Stateflow® and Stateflow® Coder

For Complex Logic and State Diagram Modeling

Modeling

Simulation

Implementation

User's Guide

Version 5

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098 Mail

For contact information about worldwide offices, see the MathWorks Web site.

Stateflow® and Stateflow Coder® User's Guide

© COPYRIGHT 1997 - 2002 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:

May 1997	First printing	
January 1999	Second printing	Revised for Stateflow 2.0 (Release 11)
September 2000	Third printing	Revised for Stateflow 4.0 (Release 12)
June 2001	Fourth printing	Revised for Stateflow 4.1 (Release 12.1)
October 2001	Online only	Revised for Stateflow 4.2 (Release 12.1+)
July 2002	Fifth printing	Revised for Stateflow 5.0 (Release 13)

Contents

Preface

System Requirements . . . . . **xxx**

Using Stateflow on a Laptop Computer . . . . . **xxxi**

Related Products . . . . . **xxxii**

Using This Guide . . . . . **xxxiii**

Typographical Conventions . . . . . **xxxv**

Installing Stateflow . . . . . **xxxvi**

1 Introduction to Stateflow

What Is Stateflow? **1-2**
Stateflow is Part of Simulink **1-2**
Stateflow is a Finite State Machine **1-3**
Stateflow Adds Flow Diagrams to the State Machine **1-4**
Stateflow Simulates its State Machine **1-5**
Stateflow Generates Code **1-7**

Build a Stateflow Model **1-8**
Create a Simulink Model **1-8**
Create a Stateflow Diagram **1-10**
Define Input Events **1-15**
Define Input Data **1-17**
Define the Stateflow Interface **1-18**
Define Simulink Parameters **1-20**
Parse the Stateflow Diagram **1-21**

Run a Simulation 1-22
Debug the Model During Simulation 1-25
Save the Model 1-26

More About Stateflow 1-27
Examples of Stateflow Applications 1-27
Stateflow Works with Simulink and Real-Time Workshop . . . 1-27
Stateflow and Simulink Design Approaches 1-27

2 How Stateflow Works

Finite State Machine Concepts 2-2
What Is a Finite State Machine? 2-2
Finite State Machine Representations 2-2
Stateflow Representations 2-3
Notation 2-3
Semantics 2-4
References 2-4

Stateflow and Simulink 2-5
The Simulink Model and the Stateflow Machine 2-5
Stateflow Data Dictionary of Objects 2-6
Defining Stateflow Interfaces to Simulink 2-7

Stateflow Graphical Components 2-9

Stateflow Diagram Objects 2-10
Graphical Objects Example Diagram 2-11
States 2-11
Transitions 2-13
Default Transitions 2-14
Events 2-15
Data 2-15
Conditions 2-16
History Junction 2-17
Actions 2-18

Connective Junctions 2-19

Stateflow Hierarchy of Objects 2-21

Exploring a Real-World Stateflow Application 2-22
Overview of the “fuel rate controller” Model 2-22
Control Logic of the “fuel rate controller” Model 2-25
Simulating the “fuel rate controller” Model 2-28

3 Stateflow Notation

Overview of Stateflow Objects 3-3
Graphical Objects 3-3
Nongraphical Objects 3-4
The Data Dictionary 3-5
How Hierarchy Is Represented 3-5

States 3-8
What Is a State? 3-8
State Decomposition 3-9
State Label Notation 3-10

Transitions 3-14
What Is a Transition? 3-14
Transition Label Notation 3-15
Valid Transitions 3-17

Transition Connections 3-18
Transitions to and from Exclusive (OR) States 3-18
Transitions to and from Junctions 3-19
Transitions to and from Exclusive (OR) Superstates 3-19
Transitions to and from Substates 3-20
Self-Loop Transitions 3-22
Inner Transitions 3-22

Default Transitions 3-26

What Is a Default Transition? 3-26
Drawing Default Transitions 3-26
Labeling Default Transitions 3-27
Default Transition Examples 3-27

Connective Junctions 3-31
What Is a Connective Junction? 3-31
Flow Diagram Notation with Connective Junctions 3-32

History Junctions 3-38
What Is a History Junction? 3-38
History Junctions and Inner Transitions 3-39

Boxes 3-40

Graphical Functions 3-41

Action Language 3-42
What Is Action Language? 3-42
Action Language Categories Example 3-43
Action Language Notation 3-43
Action Language Semantics 3-44

4 Stateflow Semantics

Executing an Event 4-3
Sources for Stateflow Events 4-3
Processing Events 4-4

Executing a Chart 4-5
Executing an Inactive Chart 4-5
Executing an Active Chart 4-5

Executing a Transition 4-6
Transition Flow Graph Types 4-6
Executing a Set of Flow Graphs 4-7

Ordering Single Source Transitions 4-8

Executing a State 4-13
Entering a State 4-13
Executing an Active State 4-15
Exiting an Active State 4-15
State Execution Example 4-16

Early Return Logic for Event Broadcasts 4-18

Semantic Examples 4-21

Transitions to and from Exclusive (OR) States Examples 4-23
Label Format for a State-to-State Transition Example 4-23
Transitioning from State to State with Events Example 4-24
Transitioning from a Substate to a Substate with Events Example 4-27

Condition Action Examples 4-29
Condition Action Example 4-29
Condition and Transition Actions Example 4-31
Condition Actions in For Loop Construct Example 4-32
Condition Actions to Broadcast Events to Parallel (AND) States Example 4-32
Cyclic Behavior to Avoid with Condition Actions Example . . . 4-33

Default Transition Examples 4-35
Default Transition in Exclusive (OR) Decomposition Example 4-35
Default Transition to a Junction Example 4-36
Default Transition and a History Junction Example 4-37
Labeled Default Transitions Example 4-39

Inner Transition Examples 4-41
Processing Events with an Inner Transition in an Exclusive (OR) State Example 4-41
Processing Events with an Inner Transition to a Connective Junction Example 4-45
Inner Transition to a History Junction Example 4-48

Connective Junction Examples **4-50**
Label Format for Transition Segments Example **4-50**
If-Then-Else Decision Construct Example **4-51**
Self-Loop Transition Example **4-53**
For Loop Construct Example **4-54**
Flow Diagram Notation Example **4-55**
Transitions from a Common Source
to Multiple Destinations Example **4-57**
Transitions from Multiple Sources
to a Common Destination Example **4-59**
Transitions from a Source to a Destination Based
on a Common Event Example **4-60**
Backtracking Behavior in Flow Graphs Example **4-61**

Event Actions in a Superstate Example **4-63**

Parallel (AND) State Examples **4-65**
Event Broadcast State Action Example **4-65**
Event Broadcast Transition Action
with a Nested Event Broadcast Example **4-69**
Event Broadcast Condition Action Example **4-72**

Directed Event Broadcasting Examples **4-77**
Directed Event Broadcast Using send Example **4-77**
Directed Event Broadcasting
Using Qualified Event Names Example **4-79**

5 Working with Charts

Creating a Stateflow Chart **5-3**

Using the Stateflow Editor **5-6**
Stateflow Diagram Editor Window **5-6**
Drawing Objects **5-8**
Displaying the Context Menu for Objects **5-10**
Specifying Colors and Fonts **5-10**

Selecting and Deselecting Objects 5-13
Cutting and Pasting Objects 5-13
Copying Objects 5-14
Editing Object Labels 5-14
Viewing Data and Events from the Editor 5-15
Zooming a Diagram 5-15
Undoing and Redoing Editor Operations 5-16

Using States in Stateflow Charts 5-18
Creating a State 5-18
Moving and Resizing States 5-19
Creating Substates 5-20
Grouping States 5-20
Specifying Substate Decomposition 5-21
Specifying Activation Order for Parallel States 5-21
Changing State Properties 5-22
Labeling States 5-23
Outputting State Activity to Simulink 5-26

Using Transitions in Stateflow Charts 5-28
Creating a Transition 5-28
Creating Straight Transitions 5-29
Labeling Transitions 5-30
Moving Transitions 5-31
Changing Transition Arrowhead Size 5-33
Creating Self-Loop Transitions 5-33
Creating Default Transitions 5-34
Setting Smart Behavior in Transitions 5-34
What Smart Transitions Do 5-35
What Nonsmart Transitions Do 5-42
Changing Transition Properties 5-44

Using Boxes in Stateflow Charts 5-47

Using Graphical Functions in Stateflow Charts 5-48
Creating a Graphical Function 5-48
Calling Graphical Functions 5-53
Exporting Graphical Functions 5-53
Specifying Graphical Function Properties 5-54

Using Junctions in Stateflow Charts **5-57**
Creating a Junction **5-57**
Changing Size **5-58**
Moving a Junction **5-58**
Editing Junction Properties **5-58**

Using Notes in Stateflow Charts **5-61**
Creating Notes **5-61**
Editing Existing Notes **5-62**
Changing Note Font and Color **5-62**
Moving Notes **5-63**
Deleting Notes **5-63**

Using Subcharts in Stateflow Charts **5-64**
What Is a Subchart? **5-64**
Creating a Subchart **5-66**
Manipulating Subcharts as Objects **5-68**
Opening a Subchart **5-68**
Editing a Subchart **5-69**
Navigating Subcharts **5-69**

Using Supertransitions in Stateflow Charts **5-71**
What Is a Supertransition? **5-71**
Drawing a Supertransition **5-72**
Labeling Supertransitions **5-77**

Specifying Chart Properties **5-78**

Checking the Chart for Errors **5-83**

Creating Chart Libraries **5-84**

Printing and Reporting on Charts **5-85**
Printing and Reporting on Stateflow Charts **5-85**
Generating a Model Report in Stateflow **5-87**
Printing the Current Stateflow Diagram **5-90**
Printing a Stateflow Book **5-90**

6 Defining Events and Data

Defining Events **6-2**
Adding Events to the Data Dictionary **6-2**
Setting Event Properties **6-4**
Defining Local Events **6-8**
Defining Input Events **6-8**
Defining Output Events **6-9**
Exporting Events **6-9**
Importing Events **6-10**
Specifying Trigger Types **6-11**
Implicit Events **6-12**

Defining Data **6-14**
Adding Data to the Data Dictionary **6-14**
Setting Data Properties **6-16**
Defining Data Arrays **6-24**
Defining Input Data **6-26**
Defining Output Data **6-27**
Associating Ports with Data **6-27**
Defining Temporary Data **6-28**
Exporting Data **6-28**
Importing Data **6-29**

7 Action Language

Types of Action Language **7-3**

Keywords in Action Language **7-4**
State Action Type Keywords **7-4**
Implicit Event Keywords **7-5**
Special Function Keywords **7-6**

Operations **7-7**
Binary and Bitwise Operations **7-7**

Unary Operations 7-10
Unary Actions 7-11
Assignment Operations 7-11
Pointer and Address Operations 7-12
Typecast Operations 7-13

Fixed-Point Data 7-14
Fixed-Point Arithmetic 7-15
How Stateflow Implements Fixed-Point Data 7-17
Specifying Fixed-Point Data in Stateflow 7-19
Tips and Tricks for Using Fixed-Point Data in Stateflow 7-20
Offline and Online Conversions of Fixed-Point Data 7-22
Fixed-Point Context-Sensitive Constants 7-23
Supported Operations with Fixed-Point Operands 7-24
Promotion Rules for Fixed-Point Operations 7-27
Assignment (=, :=) and Cast Operations 7-32
Overflow Detection for Fixed-Point Types 7-37
Sharing Fixed-Point Data with Simulink 7-37
Fixed-Point “Bang-Bang Control” Example 7-38

Calling C Functions 7-42
Calling C Library Functions 7-42
Calling min and max Functions 7-42
Calling User-Written C Code Functions 7-43

MATLAB Functions and Data 7-46
ml Namespace Operator 7-46
ml Function 7-47
ml Expressions 7-49
Which ml Should I Use? 7-50
ml Data Type 7-51
Inferring Return Size for ml Expressions 7-53

Data and Event Arguments 7-58

Arrays in Action Language 7-60
Array Notation 7-60
Arrays and Custom Code 7-61

Event Broadcasting **7-62**
Broadcasting Events **7-62**
Directed Event Broadcasting **7-64**

Condition Statements **7-67**

Using Temporal Logic **7-68**
Rules for Using Temporal Logic Operators **7-68**
after Temporal Logic Operator **7-70**
before Temporal Logic Operator **7-71**
at Temporal Logic Operator **7-72**
every Temporal Logic Operator **7-73**
Conditional and Event Notation **7-74**
Temporal Logic Events **7-74**

Special Symbols **7-76**

8 Defining Stateflow Interfaces

Overview Stateflow Interfaces **8-2**
Stateflow Interfaces **8-2**
Typical Tasks to Define Stateflow Interfaces **8-3**
Where to Find More Information on Events and Data **8-3**

Setting the Stateflow Block Update Method **8-4**
Stateflow Block Update Methods **8-4**

Adding Input or Output Data and Events to Charts **8-6**
Adding Input Events from Simulink **8-6**
Adding Output Events to Simulink **8-7**
Adding Input Data from Simulink **8-8**
Adding Output Data to Simulink **8-9**

Implementing Interfaces in Stateflow to Simulink **8-11**
Defining a Triggered Stateflow Block **8-11**
Defining a Sampled Stateflow Block **8-12**

Defining an Inherited Stateflow Block 8-13
Defining a Continuous Stateflow Block 8-14
Defining Function Call Output Events 8-16
Defining Edge-Triggered Output Events 8-19

MATLAB Workspace Interfaces 8-22
Examining the MATLAB Workspace in MATLAB 8-22
Interfacing the MATLAB Workspace in Stateflow 8-22

Interface to External Sources 8-23
Exported Events 8-23
Imported Events 8-25
Exported Data 8-27
Imported Data 8-29

9 Truth Tables

Introduction to Truth Tables 9-2
What Are Truth Tables? 9-2
Using Truth Tables in Stateflow 9-4
Your First Truth Table 9-7

Using Truth Tables 9-14
Why Use a Truth Table? 9-14
Calling Truth Table Functions in Stateflow 9-16

Specifying Stateflow Truth Tables 9-18
How to Create a Truth Table 9-18
Editing a Truth Table 9-20
Entering Conditions 9-23
Entering Decision Outcomes 9-25
Entering Actions for Decision Outcomes 9-26
Using Stateflow Data and Events in Truth Tables 9-32
Specifying Properties for a Truth Table 9-34

Edit Operations in a Truth Table 9-37

Undoing and Redoing Truth Table Editor Operations 9-37
Adding Rows and Columns 9-38
Editing Individual Cells 9-39
Editing Rows and Columns 9-41
Optional Single Table Display 9-44
Searching and Replacing in Truth Tables 9-45
Making Printed and Online Copies of Truth Tables 9-47

Error Checking for Truth Tables 9-48
When Stateflow Checks Truth Tables for Errors 9-48
Errors Detected During Error Checking 9-49
Warnings Detected During Error Checking 9-51
Over- and Underspecified Truth Tables 9-52

Debugging Truth Tables 9-56
How to Debug a Truth Table During Simulation 9-56
Debugging the check_temp Truth Table 9-57
Creating a Model for the check_temp Truth Table 9-60

Model Coverage for Truth Tables 9-64
Example Model Coverage Report 9-64

How Stateflow Realizes Truth Tables 9-68
When Stateflow Generates Truth Table Functions 9-68
How to See the Generated Graphical Function 9-68
How Stateflow Generates Truth Tables 9-69

10 Exploring and Searching Charts

Overview of the Stateflow Machine 10-2

The Stateflow Explorer Tool 10-3
Opening Stateflow Explorer 10-3
Explorer Main Window 10-4
Objects in the Explorer 10-5
Objects and Properties in the Contents Pane 10-5

Targets in the Explorer 10-7

Stateflow Explorer Operations 10-8
Opening a New or Existing Simulink Model 10-9
Editing States or Charts 10-9
Setting Properties for Data, Events, and Targets 10-9
Moving and Copying Events, Data, and Targets 10-10
Changing the Index Order of Inputs and Outputs 10-11
Deleting Events, Data, and Targets 10-12
Setting Properties for the Stateflow Machine 10-12
Transferring Properties Between Objects 10-14

The Stateflow Search & Replace Tool 10-16
Opening the Search & Replace Tool 10-16
Using Different Search Types 10-19
Specify the Search Scope 10-21
Using the Search Button and View Area 10-23
Specifying the Replacement Text 10-26
Using the Replace Buttons 10-27
Search and Replace Messages 10-28
The Search & Replace Tool Is a Product of Stateflow 10-30

The Stateflow Finder Tool 10-31
Opening Stateflow Finder 10-31
Using Stateflow Finder 10-32
Finder Display Area 10-35

11 Building Targets

Overview of Stateflow Targets 11-3
Target Types 11-3
Building a Target 11-3
How Stateflow Builds Targets 11-5

Setting Up the Target Compiler 11-6

Configuring a Target **11-7**
Adding a Target to a Stateflow Machine **11-7**
Accessing the Target Builder Dialog for a Target **11-9**
Specifying Build Options for a Target **11-10**
Specifying Code Generation Options **11-11**

Integrating Custom Code with Stateflow Diagrams **11-20**
Specifying Custom Code Options **11-20**
Specifying Path Names in Custom Code Options **11-22**
Calling Graphical Functions from Custom Code **11-23**

Starting the Build **11-25**
Starting a Simulation Target Build **11-25**
Starting an RTW Target Build **11-25**

Parsing Stateflow Diagrams **11-27**
Calling the Stateflow Parser **11-27**
Parsing Diagram Example **11-28**

Resolving Event, Data, and Function Symbols **11-32**

Error Messages **11-34**
Parser Error Messages **11-34**
Code Generation Error Messages **11-35**
Compilation Error Messages **11-36**

Generated Files **11-37**
DLL Files **11-37**
Code Files **11-38**
Makefiles **11-39**

12 Debugging and Testing

Overview of the Stateflow Debugger **12-2**
Typical Debugging Tasks **12-2**
Including Error Checking in the Target Build **12-3**

Breakpoints 12-3
Run-Time Debugging 12-4

Stateflow Debugger User Interface 12-5
Debugger Main Window 12-5
Status Display Area 12-7
Breakpoint Controls 12-7
Debugger Action Control Buttons 12-7
Error Checking Options 12-8
Animation Controls 12-9
Display Controls 12-9

Debugging Run-Time Errors Example 12-11
Create the Model and Stateflow Diagram 12-11
Define the sfun Target 12-13
Invoke the Debugger and Choose Debugging Options 12-13
Start the Simulation 12-13
Debug the Simulation Execution 12-14
Resolve Run-Time Error and Repeat 12-14
Debugged Stateflow Diagram 12-14

Debugging State Inconsistencies 12-16
Causes of State Inconsistency 12-16
Detecting State Inconsistency 12-16
State Inconsistency Example 12-17

Debugging Conflicting Transitions 12-18
Detecting Conflicting Transitions 12-18
Conflicting Transition Example 12-18

Debugging Data Range Violations 12-20
Detecting Data Range Violations 12-20
Data Range Violation Example 12-20

Debugging Cyclic Behavior 12-22
Cyclic Behavior Example 12-23
Flow Cyclic Behavior Not Detected Example 12-24
Noncyclic Behavior Flagged as a Cyclic Example 12-25

Stateflow Chart Model Coverage **12-26**
Making Model Coverage Reports **12-26**
Specifying Coverage Report Settings **12-27**
Cyclomatic Complexity **12-27**
Decision Coverage **12-28**
Condition Coverage **12-32**
MCDC Coverage **12-33**
Coverage Reports for Stateflow Charts **12-33**
Summary Report Section **12-34**
Details Sections **12-35**
Chart as Subsystem Report Section **12-36**
Chart as Superstate Report Section **12-37**
State Report Section **12-38**
Transition Report Section **12-40**

13 Stateflow API

Overview of the Stateflow API **13-3**
What Is the Stateflow API? **13-3**
Stateflow API Object Hierarchy **13-4**
Getting a Handle on Stateflow API Objects **13-6**
Using API Object Properties and Methods **13-7**
API References to Properties and Methods **13-7**

Quick Start for the Stateflow API **13-9**
Create a New Model and Chart **13-9**
Access the Machine Object **13-9**
Access the Chart Object **13-10**
Create New Objects in the Chart **13-11**

Accessing the Properties and Methods of Objects **13-17**
Naming Conventions for Properties and Methods **13-17**
Using Dot Notation with Properties and Methods **13-17**
Using Function Notation with Methods **13-18**

Displaying Properties and Methods **13-19**

Displaying the Names of Properties 13-19
Displaying the Names of Methods 13-20
Displaying Property Subproperties 13-20
Displaying Enumerated Values for Properties 13-21

Creating and Destroying API Objects 13-22
Creating Stateflow Objects 13-22
Establishing an Object's Parent (Container) 13-24
Destroying Stateflow Objects 13-25

Accessing Existing Stateflow Objects 13-26
Finding Objects 13-26
Finding Objects at Different Levels of Containment 13-27
Getting and Setting the Properties of Objects 13-28

Copying Objects 13-29
Accessing the Clipboard Object 13-29
copy Method Limitations 13-30
Copying by Grouping (Recommended) 13-30
Copying Objects Individually 13-31

Using the Editor Object 13-33
Accessing the Editor Object 13-33
Changing the Stateflow Display 13-33

Entering Multiline Labels 13-34

Creating Default Transitions 13-35

Making Supertransitions 13-36

Creating a MATLAB Script of API Commands 13-38

14 API Properties and Methods by Use

Reference Table Column Descriptions 14-2

Categories of Use 14-3

Structural Properties 14-5

Structural Methods 14-12

Behavioral Properties 14-13

Behavioral Methods 14-19

Deployment Properties 14-20

Deployment Methods 14-25

Utility and Convenience Properties 14-26

Utility and Convenience Methods 14-28

Graphical Properties 14-30

Graphical Methods 14-36

15 API Properties and Methods by Object

Reference Table Columns 15-3

Objectless Methods 15-4

Constructor Methods 15-5

Editor Properties 15-6

Editor Methods 15-7

Clipboard Methods 15-8

All Object Methods **15-9**

Root Methods **15-10**

Machine Properties **15-11**

Machine Methods **15-14**

Chart Properties **15-16**

Chart Methods **15-22**

State Properties **15-24**

State Methods **15-27**

Box Properties **15-29**

Box Methods **15-31**

Function Properties **15-33**

Function Methods **15-35**

Truth Table Properties **15-37**

Truth Table Methods **15-40**

Note Properties **15-41**

Note Methods **15-43**

Transition Properties **15-44**

Transition Methods **15-47**

Junction Properties **15-48**

Junction Methods **15-49**

Data Properties **15-50**

Data Methods **15-55**

Event Properties **15-56**

Event Methods **15-58**

Target Properties **15-59**

Target Methods **15-63**

16 **API Methods Reference**

Description of Method Reference Fields **16-2**

Methods — Alphabetical List **16-3**

A **Glossary**

Index

Preface

The Preface provides you with preliminary information on installing Stateflow and understanding this user's guide. The sections in this chapter are as follows:

System Requirements (p. xxx)	Lists the operating systems compatible with this version of MATLAB and Stateflow.
Using Stateflow on a Laptop Computer (p. xxxi)	Sets the Windows graphics palette for a laptop computer to operate more acceptably with this version of MATLAB and Stateflow.
Related Products (p. xxxii)	Lists other MATLAB products that are relevant to the kinds of tasks you can perform with Stateflow.
Using This Guide (p. xxxiii)	Provides a Quick Reference Guide for navigating the Stateflow User's Guide.
Typographical Conventions (p. xxxv)	Lists typographical conventions used throughout the Stateflow User's Guide.
Installing Stateflow (p. xxxvi)	Gives you special information you need before installing Stateflow.

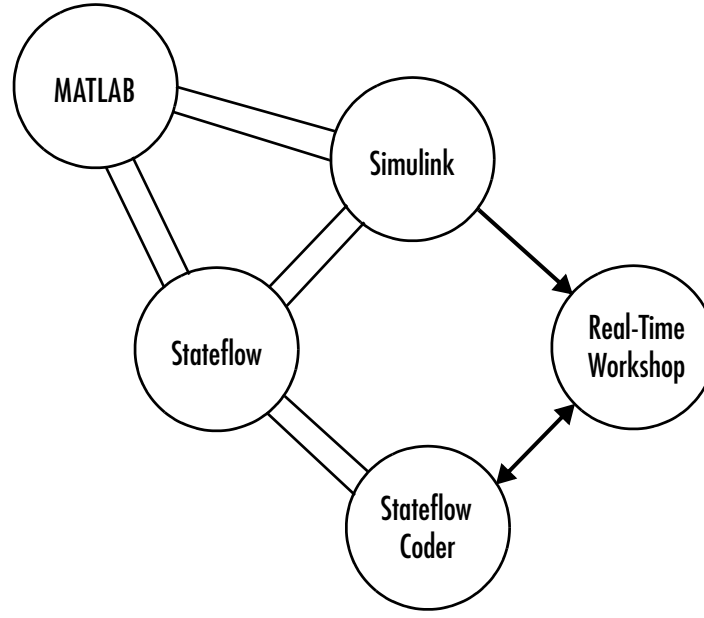
System Requirements

Stateflow® is a multiplatform product that runs on Microsoft Windows and UNIX systems.

Stateflow 5 requires the installation of the following software:

- MATLAB® 6.5 (Release13)
- Simulink® 5
- C or C++ compiler for generating code from a Stateflow model

  See “Setting Up the Target Compiler” on page 11-6 for more information.
- Real-Time Workshop® Version 5 (Release 13) for generating code for the Simulink elements of a Stateflow model



Using Stateflow on a Laptop Computer

If you plan to run the Microsoft Windows version of Stateflow on a laptop computer, you should configure the Windows color palette to use more than 256 colors. Otherwise, you can experience unacceptably slow performance.

To set the Windows graphics palette:

1. Click the right mouse button on the Windows desktop to display the desktop menu.
2. Select **Properties** from the desktop menu to display the Windows **Display Properties** dialog.
3. Select the **Settings** panel on the **Display Properties** dialog.
4. Choose a setting that is more than 256 colors from the **Color Palette** colors list.
5. Select **OK** to apply the new setting and dismiss the **Display Properties** dialog.

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with Stateflow.

For more information about any of these products, see one of the following:

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site at `www.mathworks.com`; see the "products" section
- The Stateflow Web site at `www.mathworks.com/products/stateflow`

The following products include functions that extend the capabilities of Stateflow.

Product	Description
MATLAB	The Language of Technical Computing
Real-Time Workshop	Generate C code from Simulink models
Simulink	Design and simulate continuous- and discrete-time systems
Simulink Report Generator	Automatically generate documentation for Simulink and Stateflow models
Stateflow Coder	Generate C code from Stateflow charts

Using This Guide

Use the following synopses for the chapters in the Stateflow User's Guide to help you find the information that you need.

- Chapter 1, "Introduction to Stateflow" — Gives you a quick start at learning the fundamentals of creating Stateflow diagrams.
- Chapter 2, "How Stateflow Works" — Introduces the fundamental concepts that are the foundation of Stateflow.
- Chapter 3, "Stateflow Notation" — Explains the graphical and nongraphical symbolism used to create Stateflow diagrams.
- Chapter 4, "Stateflow Semantics" — Explains how Stateflow diagram notation is interpreted and implemented into Stateflow model behavior.
- Chapter 5, "Working with Charts" — Teaches you how to create and fill Stateflow charts with Stateflow diagrams.
- Chapter 6, "Defining Events and Data" — Describes the nongraphical objects that are essential to completing and defining interfaces to the Stateflow diagram.
- Chapter 7, "Action Language" — Describes the allowable content (statements, function calls, etc.) in Stateflow action language.
- Chapter 8, "Defining Stateflow Interfaces" — Describes how to create interfaces between a chart block and other blocks in a Simulink model.
- Chapter 9, "Truth Tables" — Shows you how to fill in truth tables that Stateflow uses to generate functions with the logical behavior you specify.
- Chapter 10, "Exploring and Searching Charts" — Tells you how to find and replace objects in Stateflow charts.
- Chapter 11, "Building Targets" — Tells you how to configure Stateflow to generate code.
- Chapter 12, "Debugging and Testing" — Tells you how to debug a Stateflow model.
- Chapter 13, "Stateflow API" — Introduces you to the Stateflow API (Application Programming Interface) and gives you a quick start to using it.
- Chapter 14, "API Properties and Methods by Use" — Provides reference information on the properties and methods of the Stateflow API, categorized by types of use.

- Chapter 15, “API Properties and Methods by Object” — Provides reference information on the properties and methods of the Stateflow API categorized by Stateflow object types.
- Chapter 16, “API Methods Reference” — Provides reference information on Stateflow API methods including their calls, syntax, arguments, and returns.
- “Glossary” — Gives you definitions of key terms and concepts in Stateflow.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, user input, items in drop-down lists	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'whole')`
Mathematical expressions	*Italic* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace` font	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italic*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italic`*	`sysc = d2c(sysd,'method')`

Installing Stateflow

Your platform-specific MATLAB Installation documentation provides essentially all the information you need to install Stateflow.

Prior to installing Stateflow, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you are permitted to install and use.

Stateflow and Stateflow Coder have certain product prerequisites that must be met for proper installation and execution.

Licensed Product	Prerequisite Products	Additional Information
Simulink 5	MATLAB 6.5 (Release 13)	Allows installation of Simulink and Stateflow in Demo mode.
Stateflow	Simulink 5	
Stateflow Coder	Stateflow	

If you experience installation difficulties and have Web access, look for license manager and installation information on the MathWorks support page at `www.mathworks.com/support`.

1

Introduction to Stateflow

This chapter introduces you to Stateflow by helping you construct a Stateflow model and execute it. The sections in this chapter are as follows:

What Is Stateflow? (p. 1-2)	This section helps you understand the nature of Stateflow and what it is capable of doing.
Build a Stateflow Model (p. 1-8)	This section provides you with a tutorial guide that gets you started quickly in using Stateflow.

What Is Stateflow?

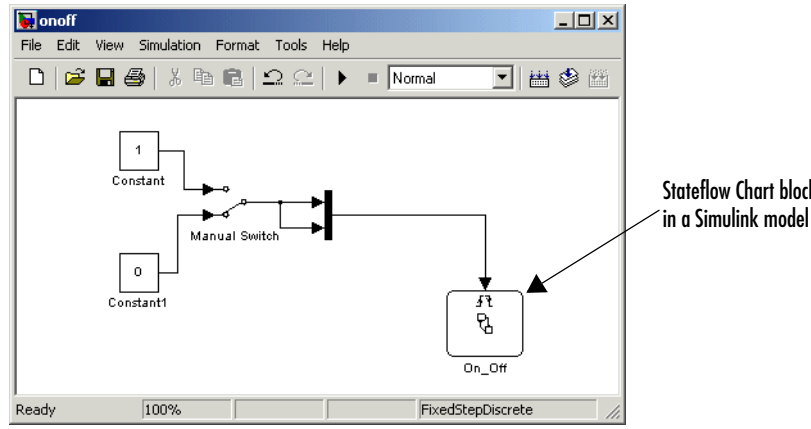
Stateflow is a graphical design and development tool for control and supervisory logic used in conjunction with Simulink. It provides clear, concise descriptions of complex system behavior using finite state machine theory, flow diagram notations, and state-transition diagrams all in the same Stateflow diagram as described in the following topics:

- "Stateflow is Part of Simulink" on page 1-2 — Stateflow integrates with its Simulink environment to model, simulate, and analyze your system.
- "Stateflow is a Finite State Machine" on page 1-3 — Stateflow visually models and simulates complex reactive control and simulation based on *finite state machine* theory. You design and develop deterministic, supervisory control systems in a graphical environment.
- "Stateflow Adds Flow Diagrams to the State Machine" on page 1-4 — Flow diagram notation creates decision-making logic such as `for` loops and `if-then-else` constructs without the use of states. In some cases, using flow diagram notation provides a closer representation of the required system logic that avoids the use of unnecessary states.
- "Stateflow Simulates its State Machine" on page 1-5 — Easily modify your design, evaluate the results, and verify the system's behavior at any stage of your design.
- "Stateflow Generates Code" on page 1-7 — Stateflow automatically generates integer, floating-point, or fixed-point code directly from your design (requires Stateflow Coder)

Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behavior.

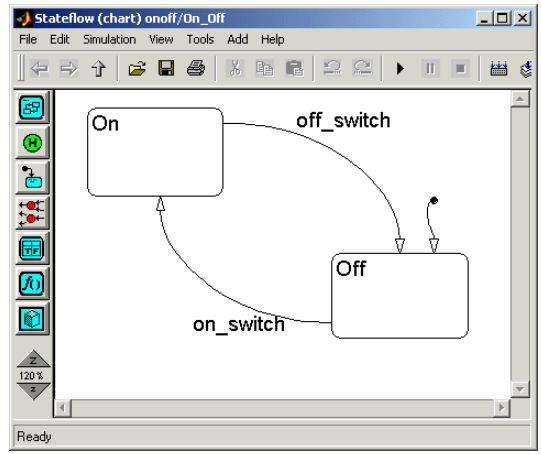
Stateflow is Part of Simulink

Stateflow is a product that is part of Simulink. In Simulink, Stateflow blocks are referred to as Stateflow Chart blocks. The following diagram shows a simple Simulink model that has a Stateflow Chart block in it:



Stateflow is a Finite State Machine

If you double-click the Stateflow block in the preceding Simulink model, its Stateflow diagram appears in the Stateflow diagram editor window.



Stateflow is an example of a *finite state machine*. A finite state machine reacts to events by changing states. The preceding example has two states: Power_on and Power_off. When you first turn the state machine on, this chart is set to execute a special transition called a *default transition* that points to the inital state, Power_off. This makes the Power_off state active. Later, when you

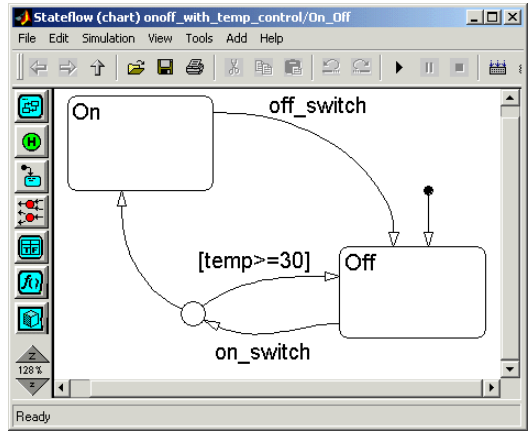
change the manual switch in Simulink from Off to On, the model sends an event (on_switch) that makes the Off state transition to the On state. This makes the Off state inactive and and On state active.

You'll get a chance to see this for yourself when you build and simulate this model in "Build a Stateflow Model" on page 1-8.

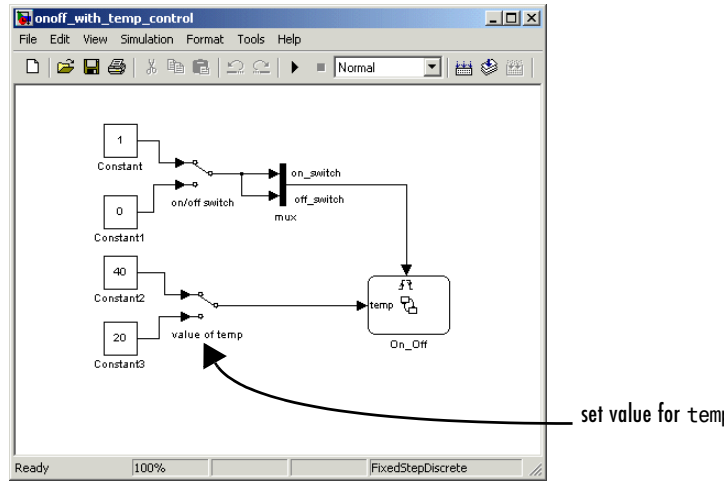
Stateflow Adds Flow Diagrams to the State Machine

While states and transitions are the basic building blocks of a state machine, Stateflow also adds the feature of flow diagrams that provide decision points in transitions. Stateflow implements flow diagrams with junctions, round objects that provide an alternative path for transitions.

In the following example, a junction has been added to the preceding diagram that checks to see if the outside temperature is less than 30 degrees. If it is, the Off state is reentered and the fan stays off. If not, the transition to the On state is taken and the fan is turned on.

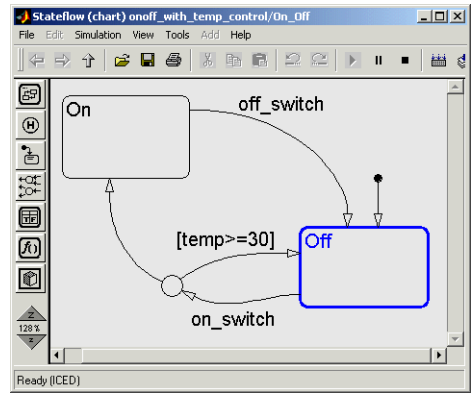


In the preceding example, outside temperature is represented by `temp`, an example of Stateflow data. Stateflow data are variables or constants that you define for Stateflow diagrams that apply locally or outside the diagram. The following diagram shows that the value for `temp` is set in Simulink, where you can change the value any time you want by switching between two constant values.

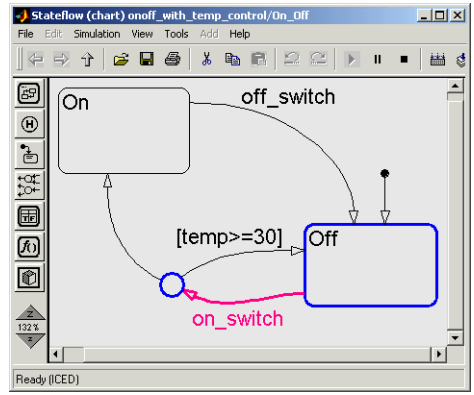


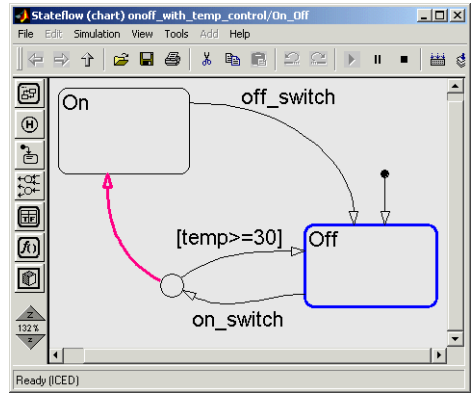
Stateflow Simulates its State Machine

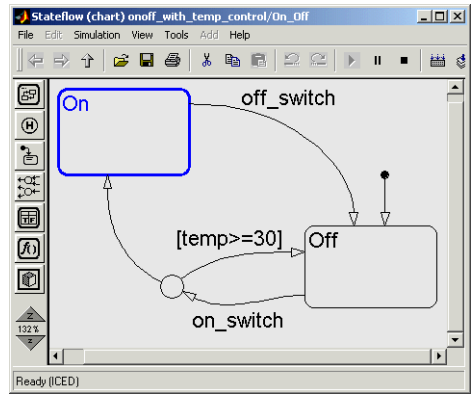
Once you finish a Simulink model with Stateflow charts, you can simulate it. This allows you to see an animated Stateflow chart while it responds to events and changes states. In the animated chart, active states are highlighted as shown in the following example:



When events occur and transitions are taken, they too are highlighted. Continuing with the previous example, the following figures shows what happens when an `on_switch` event occurs and the value of `temp` is less than 30:


Stateflow (chart) onoff_with_temp_control/On_Off
File Edit Simulation View Tools Add Help
On
off_switch
[temp>=30]
Off
on_switch
132%
Ready (ICED)


Stateflow (chart) onoff_with_temp_control/On_Off
File Edit Simulation View Tools Add Help
On
off_switch
[temp>=30]
Off
on_switch
132%
Ready (ICED)


Stateflow (chart) onoff_with_temp_control/On_Off
File Edit Simulation View Tools Add Help
On
off_switch
[temp>=30]
Off
on_switch
132%
Ready (ICED)

Stateflow Generates Code

Stateflow generates its own C-code to simulate Stateflow charts during simulation. When you generate code for simulation, this is referred to as generating a simulation target whose name is sfun.

If you have Stateflow Coder, you can generate code for applications that you build in other environments, such as an embedded environment. You can even include code of your own, custom code, that Stateflow uses to build the target with.

- With Real-Time Workshop (RTW) tool you can take code from Simulink and Stateflow and run it as an application on another environment to control a process.
- You can also generate code from Stateflow charts that you use in any way that you want. You simply generate code and use it as you wish.

Build a Stateflow Model

In this section, you use Stateflow to create, run, and debug the simple power switch model introduced to you in the previous section. The steps you take in this quick start to learning Stateflow are as follows:

1. "Create a Simulink Model" on page 1-8 — Start a model in Simulink containing a Stateflow block.
2. "Create a Stateflow Diagram" on page 1-10 — Start a Stateflow diagram for the Stateflow block in your Simulink model.
3. "Define Input Events" on page 1-15 — Define the input events from Simulink that trigger the execution of your Stateflow diagram.
4. "Define Input Data" on page 1-17 — Define the input data from Simulink that controls a transition with a condition.
5. "Define the Stateflow Interface" on page 1-18 — Add a source of triggers in the Simulink model for executing the Stateflow diagram.
6. "Define Simulink Parameters" on page 1-20 — Define operating parameters for the Simulink model during simulation.
7. "Parse the Stateflow Diagram" on page 1-21 — Check the Stateflow diagram for syntactical errors.
8. "Run a Simulation" on page 1-22 — Go through the steps for simulating your model.
9. "Debug the Model During Simulation" on page 1-25 — Tells you how you can use the Stateflow Debugger to carefully monitor all aspects of your quick start model during simulation.
10. "Save the Model" on page 1-26 — Save the model that you created and simulated in the Quick Start section.

Create a Simulink Model

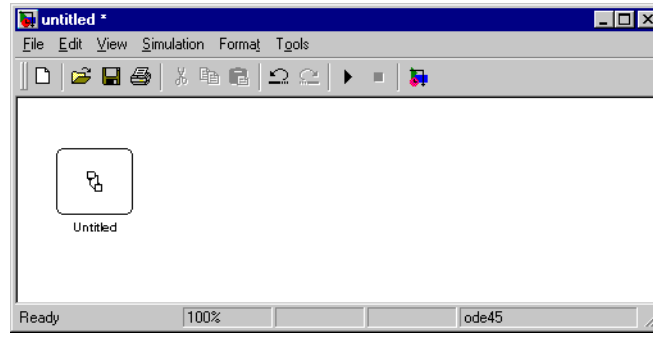
Opening the Stateflow model window is the first step toward creating a Simulink model with a Stateflow block. By default, an untitled Simulink model

with an untitled, empty Stateflow block is created for you when you open the Stateflow model window. You can either start with the default empty model or copy the untitled Stateflow block into any Simulink model to include a Stateflow diagram in an existing Simulink model.

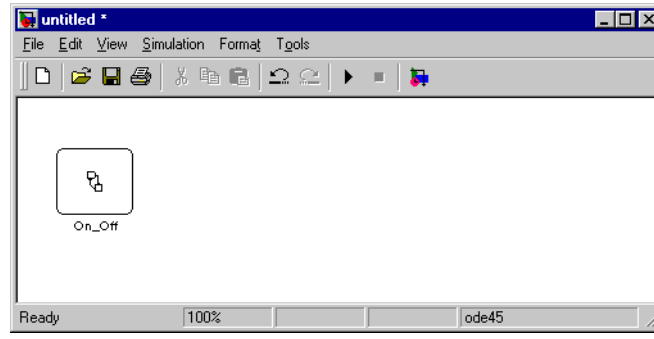
The following steps describe how to create a Simulink model with a Stateflow block, label the Stateflow block, and save the model:

1 At the MATLAB prompt enter `sfnew`.

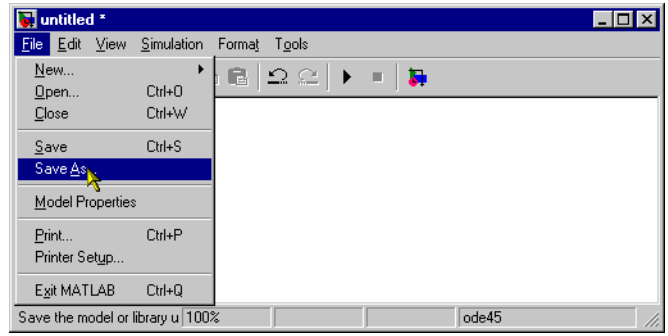
Stateflow displays the following untitled Simulink model window with an untitled Stateflow block.



2 Label the Stateflow block in the new untitled model by clicking in the text area and replacing the text `Untitled` with the text `On_Off`.



3 Save the model by choosing **Save As** from the **File** menu of the Simulink model window and entering a model title.

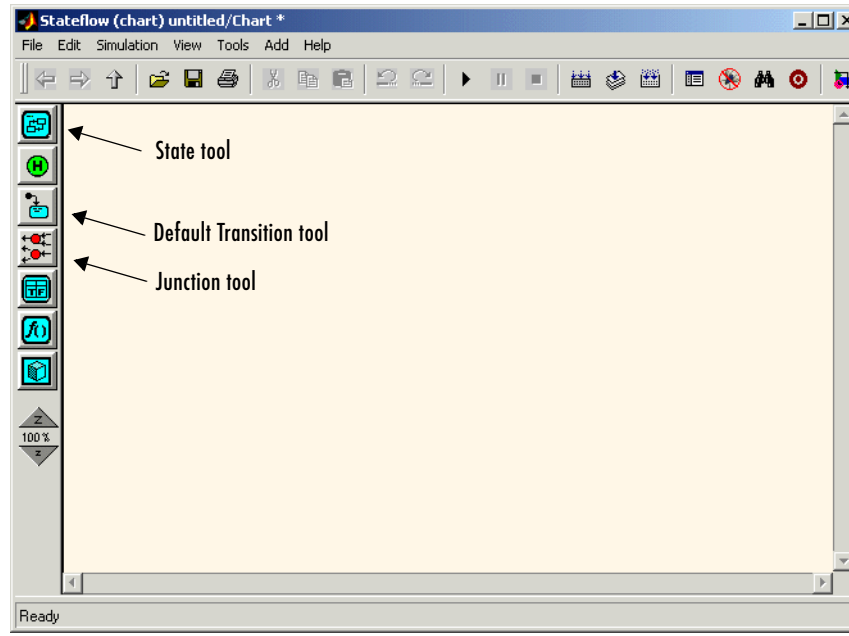


Create a Stateflow Diagram

The following steps describe how to create a simple Stateflow diagram using the graphics editor:

1 Double-click the Stateflow block in the Simulink model window to invoke the graphics editor window.

Notice the tool icons the drawing toolbar that you will soon be using:



2 Select the **State** tool button in the drawing toolbar.

3 Move the cursor into the drawing area and left-click to place the state.

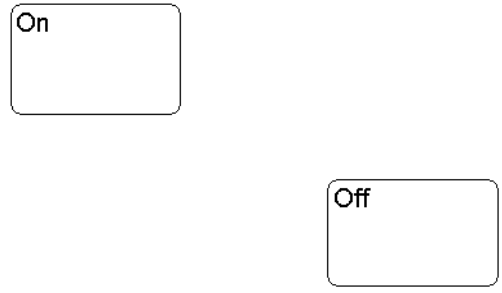
4 Position the cursor over that state, click the right mouse button, and drag to another location in the drawing area to make a copy of the state. Release the right mouse button to drop the state at that location.

5 Click the ? character within each state to enter each state label.

Label the states with the titles `On` and `Off`.

6 Deselect the state to exit the edit.

To deselect a state, click anywhere outside the state or press the **Esc** key. Your Stateflow diagram should look similar to the following:



7 Draw a transition starting from the right side of state `On` to the top of state `Off` as follows:

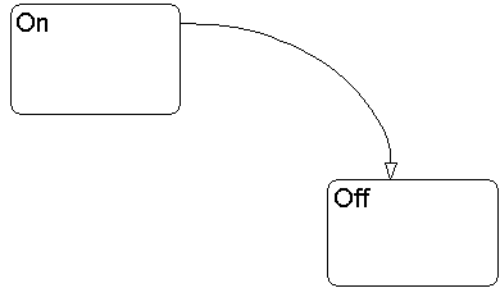
   a Place the cursor at a straight portion of the right border of the `On` state.

     Crosshairs do not appear if you place the cursor on a corner, since corners are used for resizing.

   b When the cursor changes to crosshairs, click-drag the mouse to the top border of the `Off` state.

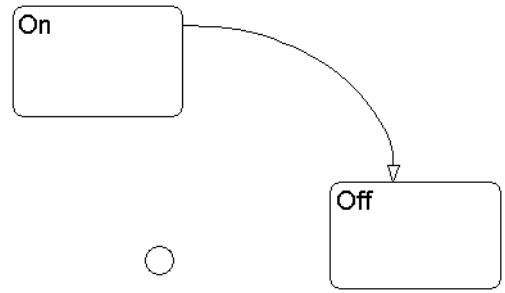
   c When the transition snaps to the border of the `Off` state, release the mouse button.

   Your Stateflow diagram now an appearance similar to the following:

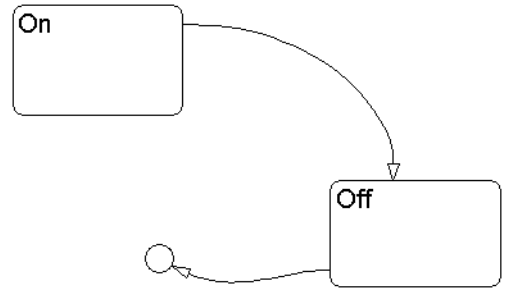


8 Select the **Junction** button in the drawing toolbar.

9 Move the cursor into the drawing area and click to place the junction as follows:

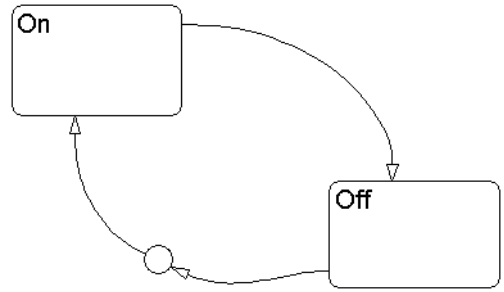


10 Draw a transition segment from the state `Off` to the junction as shown.

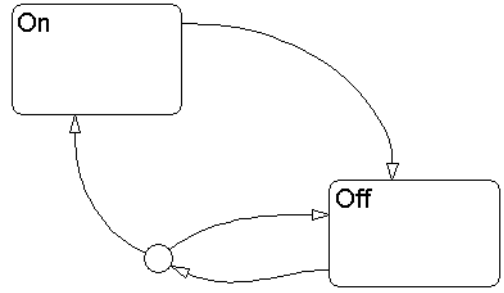


Transitions exist only between one state and another. However, transitions can be made up of transition segments that define alternate flow paths with junctions.

11 Draw a transition segment from the junction to the state `On` as shown.



12 Draw a transition segment from the junction to the state `Off` as shown.



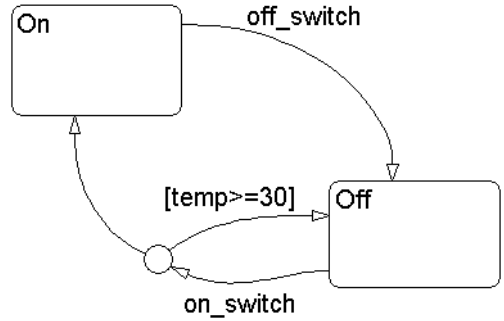
13 Label the transition from the state `On` to the state `Off` as follows:

- **a** Click the transition to select it.
- **b** Click the **?** character that appears alongside the transition to obtain a blinking cursor.
- **c** Enter the label text `off_switch`.
- **d** Press the **Escape** key to deselect the transition label and exit edit mode for this label.

14 Label the transition segment from the state `Off` to the junction with the text `on_switch`.

15 Label the transition segment from the junction to the state `Off` with the text `[temp > 30]`.

Your Stateflow diagram should now look similar to the following:



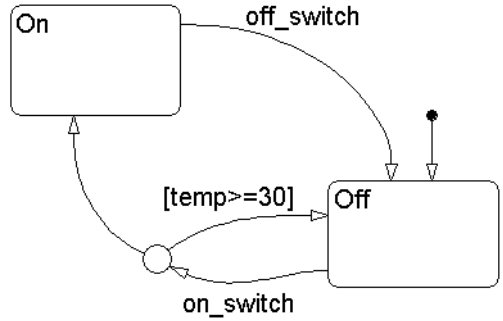
16 Add a default transition to the state Off with the following steps:

a Select the **Default Transition** tool in the drawing toolbar.

b Move the mouse cursor to the top border of the `Off` state.

c Click and release the mouse when the cursor's transition arrowhead snaps straigh up and down to the border of the `Off` state.

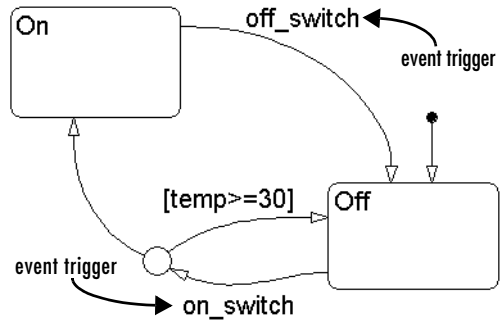
Your finished Stateflow diagram should now look similar to the following:



Note For more information on creating Stateflow diagrams using the graphics editor, see Chapter 5, "Working with Charts."

Define Input Events

Before you can simulate the Stateflow diagram that you finished in "Create a Stateflow Diagram" on page 1-10, you must define the events `on_switch` and `off_switch` that trigger the transitions between `On` state and `Off` state. These event triggers are pointed out in the finished diagram:

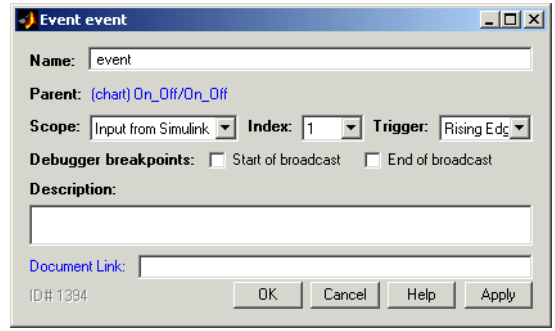


Events control the execution of your Stateflow diagram. For example, if the state `On` is active and the `on_switch` event occurs, the transition from the `On` state to the `Off` state takes place and the `Off` state becomes active.

To define these input events for your Stateflow diagram, do the following:

1. Select **Event** from the **Add** menu of the diagram editor.
2. In the resulting submenu, select **Input from Simulink**.

   The property dialog for the new event appears.



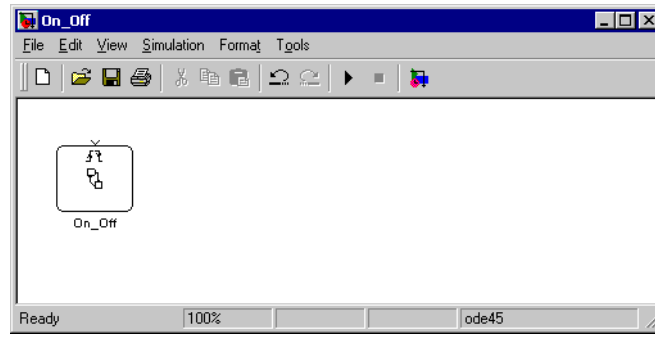
3. Enter `on_switch` in the **Name** field of the **Event properties** dialog box.
4. Select **Rising Edge** as the **Trigger** type.

   Leave all other fields with their default values.

5 Select **OK** to apply the changes and close the window.

6 Repeat steps 1 through 5 to define the event `off_switch` except enter **Falling Edge** as the **Trigger** type in step 4.

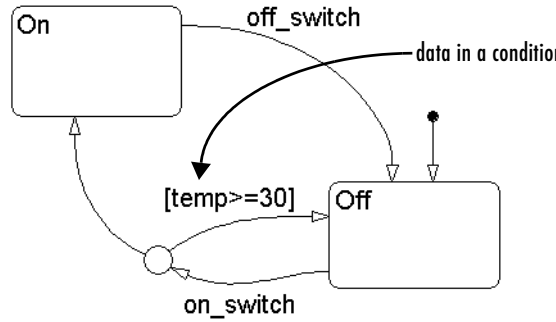
Now that you have defined the `on_switch` and `off_switch` events for your Stateflow diagram, take a look at the Stateflow block in the Simulink model.



The Stateflow block now has an input port for the events that you defined. Even though you have defined these events, you must now provide a source for these events in Simulink which you will do in “Define the Stateflow Interface” on page 1-18.

Define Input Data

Since the transition segment between the junction and the state `Off` has a condition based on the value of the Stateflow data temp, you must define temp in the diagram.

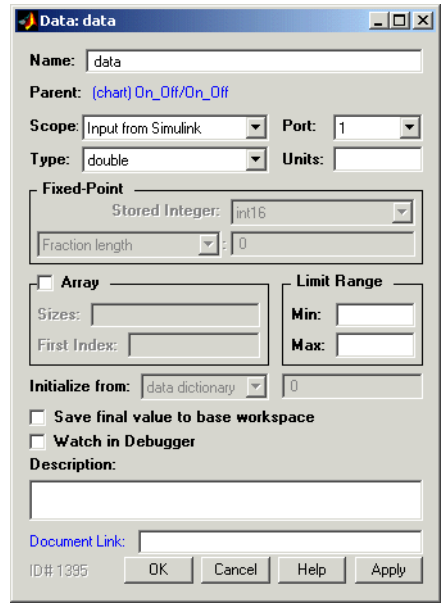


To define the input data `temp` for your Stateflow diagram, do the following:

1 Select **Data** from the **Add** menu of the diagram editor.

2 In the resulting submenu, select **Input from Simulink**.

The property dialog for the new data appears.



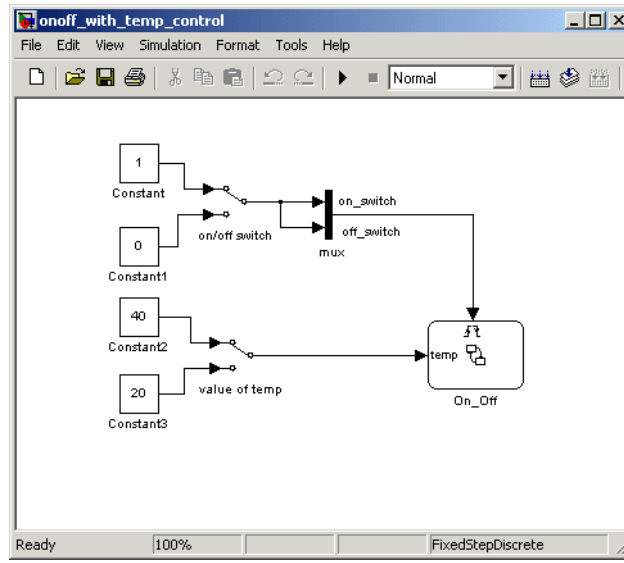
3 Enter `temp` in the **Name** field of the properties dialog.

Leave all other fields with their default values.

4 Select **OK** to apply the changes and close the window.

Define the Stateflow Interface

Now that you have defined events and data in your Stateflow diagram as **Input from Simulink**, you must now make connections in the Simulink model between other blocks and the Stateflow block to provide sources for the events and data. This model has the following final appearance:



To construct this model in Simulink using your existing chart, do the following:

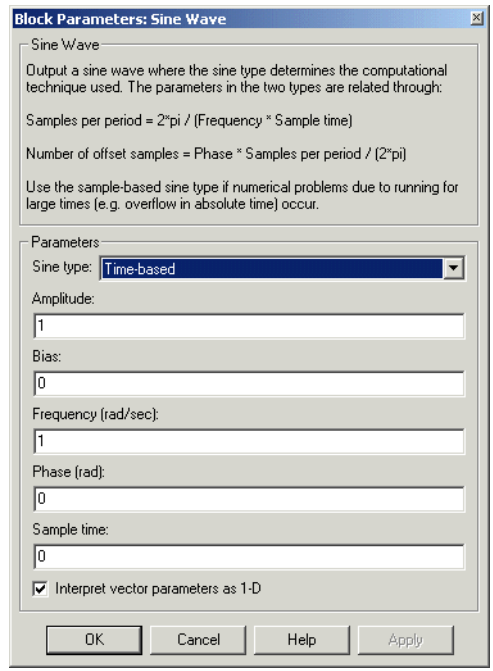
1. Add two Manual Switch blocks (located in the Simulink Sources block library).
2. Label the top switch `Switch Events`, and the bottom switch `temp value`.
3. Place four Constant blocks (located in the Simulink Sources block library) to the left of the switches.
4. Connect two Constant blocks to each switch to provide pole inputs.
5. Double-click each Constant block to set its **Constant value** parameter as shown in the previous diagram.
6. Connect the output of the temp value switch to the temp data input port on the Stateflow block.
7. Place a Mux block (located in the Simulink Sinks block library) on the output side of the `Switch Events` switch.
8. Double-click the Mux block to set its **Number of inputs** parameter to 2.

9 Connect the output of the `Switch Events` switch in parallel to both of the Mux input ports.

10 Connect the single output port of the Mux to the trigger input port of the Stateflow Chart block `On-Off`.

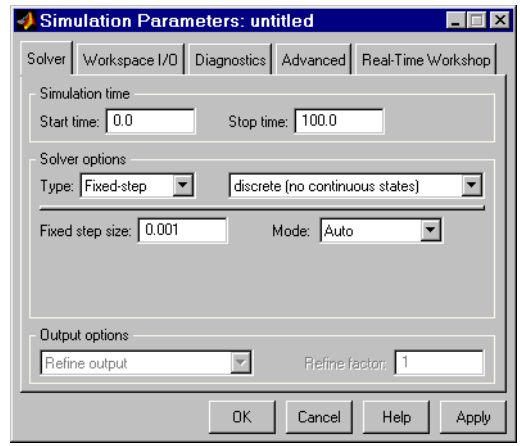
Define Simulink Parameters

1 Double-click the Sine Wave block and edit the parameters as shown in this partial dialog box example.


Block Parameters: Sine Wave
Sine Wave
Output a sine wave where the sine type determines the computational technique used. The parameters in the two types are related through:
Samples per period = 2*pi / (Frequency * Sample time)
Number of offset samples = Phase * Samples per period / (2*pi)
Use the sample-based sine type if numerical problems due to running for large times (e.g. overflow in absolute time) occur.
Parameters
Sine type: Time-based
Amplitude:
1
Bias:
0
Frequency (rad/sec):
1
Phase (rad):
0
Sample time:
0
Interpret vector parameters as 1-D
OK
Cancel
Help
Apply

Select **OK** to apply the changes and close the dialog box.

2 Choose **Simulation Parameters** from the **Simulation** menu of the Simulink model window and edit the values to match the values in this dialog box.



Select **OK** to apply the changes and close the dialog box.

3 Select **Save** from the **File** menu to save the model.

Note For more information, see Chapter 8, “Defining Stateflow Interfaces.”

Parse the Stateflow Diagram

Parsing the Stateflow diagram ensures that the notations you specified are valid and correct. To parse the Stateflow diagram, choose **Parse Diagram** from the **Tools** menu of the graphics editor. Informational messages are displayed in the MATLAB Command Window. Any error messages are displayed in red. If no messages appear with a red button, the parse operation is successful.

Note For more information, see “How Stateflow Builds Targets” on page 11-5.

Run a Simulation

These steps show how to run a simulation:

1. Double-click the `On_off` Stateflow block to display the Stateflow diagram.
2. Select **Open Simulation Target** from the graphics editor **Tools** menu.

   The **Simulation Target Builder** dialog box appears.
3. Select **Coder Options** on the **Simulation Target Builder** dialog box.

   The **Simulation Coder Options** dialog box appears.
4. Ensure that the check box to **Enable Debugging/Animation** is selected.
5. Select **OK** to apply the change and close the **Simulation Coder Options** dialog box.
6. Close the **Simulation Target Builder** dialog box.
7. Select **Debug** from the graphics editor **Tools** menu.

   The Stateflow Debugging dialog box appears.
8. Ensure that the **Enabled** radio button in the **Animation** section is selected.
9. Select **Close** to apply the change and close the window.
10. Choose **Start** from the diagram editor **Simulation** menu to start a simulation of the model and notice the following:
    - Stateflow displays code generation status messages in the MATLAB Command Window while it builds the simulation target (`sfun`).
    - Before starting the simulation, Stateflow temporarily sets the model to read-only to prevent accidental modification while the simulation is running. This is displayed as “Ready (ICED)” where the term “ICED” is an internal Stateflow designation.
    - The Stateflow diagram editor background becomes shaded, and the default transition into state Off highlights followed by the highlighting of state Off itself.

11 Toggle the `on/off switch` manual switch (double click it) between its inputs, 0 and 1.

Toggling the `on/off switch` sends `on_switch` and `off_switch` events to the Stateflow chart. These events toggle the chart's active state from `Off` to `On` and back to `Off` again.

Both the `on_switch` and `off_switch` events are defined as **Input from Simulink** events. This means that the events are input to the chart from Simulink, in this case, from a manual switch. When the input to the `on/off switch` rises from 0 to 1, this sends an `on_switch` event to the `On_Off` chart because `on_switch` is defined as a rising trigger. Similarly, when the input to the `on/off switch` falls from 1 to 0, this sends an `off_switch` event to the chart because the `off_switch` event is defined as a falling trigger.

12 Toggle the `value of temp` switch to change the value of the data `temp` from 20 to 40.

13 Toggle the `on/off switch` again and notice the difference.

Because of the different value for `temp`, the transition from the `Off` state to the `On` state no longer takes place. The transition from `Off` to `On` passes through a junction which has an alternate path back to `Off`. Because the alternate path has a condition, it gets priority. In this case, the condition evaluates to true (nonzero) and the alternate path is taken.

14 Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation.

Once the simulation stops, Stateflow resets the model to be editable.

Note Running a simulation might require setting up the tools used to build Stateflow targets. See "Setting Up the Target Compiler" on page 11-6 for more information.

Where Stateflow Generates Code for Simulation

When you simulate a Simulink model containing Stateflow charts (as you did in the previous section), Stateflow generates a Simulink S-function (`sfun`) target that enables Simulink to simulate the Stateflow blocks. The `sfun` target can be used only with Simulink.

Before generating code, Stateflow creates a directory called `sfprj` in the current MATLAB directory if the directory does not already exist. Stateflow uses the `sfprj` directory during code generation to store generated files. See the section "Generated Files" on page 11-37.

Note Do not confuse the `sfprj` directory created in the MATLAB current directory for generated files with the `sfprj` directory in the directory containing the model file (see the section "Save the Model" on page 1-26). The latter `sfprj` directory is used for storing information on the model, while the former stores generated files. However, if the MATLAB current directory is the model directory, the same `sfprj` directory (under the model directory) is used to store both model information and generated files.

If you have the Stateflow Coder, you can generate stand-alone code suitable for a particular processor. See "Building a Target" on page 11-3 for more information on code generation.

Debug the Model During Simulation

The Stateflow Debugger supports functions like single stepping, animating, and running up to a designated breakpoint and then stopping.

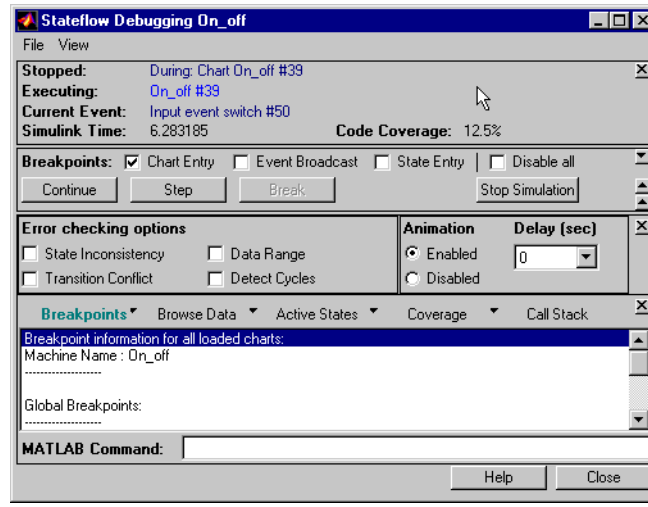
These steps show how to step through the simulation using the Debugger:

1. Display the Debugger by choosing **Debug** from the **Tools** menu of the graphics editor.

2. Select the **Breakpoints**: **Chart Entry** check box to tell the Debugger you want it to stop simulation execution when the chart is entered.

3. Select **Start** to start the simulation.

   Informational and error messages related to the S-function code generation for Stateflow blocks are displayed in the MATLAB Command Window. When the target is built, the graphics editor becomes read-only (frozen) and the Debugger window is updated and looks similar to this.



4 Select **Step** to proceed one step through the simulation. The Debugger window displays the following information:

- Where the simulation is stopped
- What is executing
- The current event
- The simulation time
- The current code coverage percentage

Watch the graphics editor window as you click the **Step** button to see each transition and state animated when it is executing. After both `Power_off` and `Power_on` have become active by stepping through the simulation, the code coverage indicates 100%.

5 Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation. Once the simulation stops, the model becomes editable.

6 Select **Close** in the Debugger window.

7 Choose **Close** from the **File** menu in the Simulink model window.

For More Information

See Chapter 12, "Debugging and Testing," for more information beyond the debugging topics in this section.

Save the Model

Save the Simulink model with your Stateflow diagram by selecting **Save Model** or **Save Model As** (for new models) from the **File** menu of the Stateflow diagram editor. You can also use the **Save** and **Save as** selections in the **File** menu of a Simulink editor window. File names for Simulink models are limited to 25 characters.

When you save your model, Stateflow creates an `sfprj` directory in the directory holding the project file. This directory is used to hold model information.

More About Stateflow

Examples of Stateflow Applications

The following types of applications benefit directly from the use of Stateflow:

- Embedded systems
  - Avionics (planes)
  - Automotive (cars)
  - Telecommunications (for example, routing algorithms)
  - Commercial (computer peripherals, appliances, and so on)
  - Programmable logic controllers (PLCs) (process control)
  - Industrial (machinery)
- Man-machine interface (MMI)
  - Graphical user interface (GUI)
- Hybrid systems
  - Air traffic control systems (digital signal processing (DSP)+control+MMI)

Stateflow Works with Simulink and Real-Time Workshop

Stateflow is used together with Simulink (and optionally with the Real-Time Workshop (RTW)) running on top of MATLAB as follows:

- MATLAB provides access to data, high-level programming, and visualization tools.
- Simulink supports development of continuous-time and discrete-time dynamic systems in a graphical block diagram environment.
- Stateflow diagrams enhance Simulink with complex event-driven control capabilities.
- Real-Time Workshop coordinates code generation from Simulink with Stateflow code generation to build an embeddable target.

Stateflow and Simulink Design Approaches

You can design a model starting with a Stateflow (control) perspective and then later build the Simulink model. You can also design a model starting from a

Simulink perspective and then later add Stateflow diagrams. You might have an existing Simulink model that would benefit if you replace Simulink logic blocks with Stateflow diagrams. The approach you use determines how, and in what sequence, you develop various parts of the model.

Stateflow and Simulink Build Simulation (sfun) Targets

The collection of all Stateflow blocks in the Simulink model is a machine. When you use Simulink together with Stateflow for simulation, Stateflow generates an S-function (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the `sfun` target within Stateflow.

Stateflow, Simulink and RTW Build RTW (rtw) Targets

Stateflow Coder generates integer or floating-point code based on the Stateflow machine. Real-Time Workshop (RTW) generates code from the Simulink portion of the model and provides a framework for running generated Stateflow code in real time. The code generated by Stateflow Coder is seamlessly incorporated into code generated by Real-Time Workshop. You might want to design a solution that targets code generated from both products for a specific platform. This generated code is specifically a Real-Time Workshop target and within Stateflow is called the RTW target.

Stateflow Builds Stand-Alone Custom Targets

Using Stateflow and Stateflow Coder you can generate code exclusively for the Stateflow machine portion of the Simulink model. This generated code is for stand-alone custom targets, which you uniquely name as something other than `sfun` and `rtw` in Stateflow.

2

How Stateflow Works

This chapter acquaints you with the concepts involved in defining a finite state machine and follows this with a look at the hierarchical organization of Stateflow objects. Later, it introduces you to a real-world Stateflow example. The sections in this chapter are as follows:

Finite State Machine Concepts (p. 2-2)	Stateflow is an example of a finite state machine. This section examines what it means to be a finite state machine and what that designation requires from Stateflow.
Stateflow and Simulink (p. 2-5)	Examines how Stateflow functions in the Simulink environment.
Stateflow Diagram Objects (p. 2-10)	This section describes most of the graphical and nongraphical objects in a Stateflow diagram along with the concepts that relate them.
Stateflow Hierarchy of Objects (p. 2-21)	Stateflow supports a containment hierarchy for both charts and states. Charts can contain states, transitions, and the other Stateflow objects. States can contain other states, transitions, and so on as if they were charts themselves. This containment hierarchy applies to all Stateflow objects except targets, which are the sole possession of the Stateflow machine.
Exploring a Real-World Stateflow Application (p. 2-22)	The modeling of a real-world fault-tolerant fuel control system demonstrates how Simulink and Stateflow can be used to efficiently model hybrid systems containing both continuous dynamics and complex logical behavior.

Finite State Machine Concepts

Stateflow is an example of a finite state machine. The following topics examine what it means to be a finite state machine and what that designation requires from Stateflow:

- "What Is a Finite State Machine?" on page 2-2 — Introduces you to the concept of a finite state machine of which Stateflow is a variant.
- "Finite State Machine Representations" on page 2-2 — Discusses traditional approaches to representing finite state machines.
- "Stateflow Representations" on page 2-3 — Discusses Stateflow's representation for its version of the finite state machine.
- "Notation" on page 2-3 — Describes how Stateflow uses its notation to represent the objects that it uses.
- "Semantics" on page 2-4 — Stateflow semantics describe how Stateflow notation is interpreted and implemented into a designed behavior.
- "References" on page 2-4 — Provides more information on finite state machine theory.

What Is a Finite State Machine?

A *finite state machine* is a representation of an event-driven (reactive) system. In an event-driven system, the system makes a transition from one state (mode) to another prescribed state, provided that the condition defining the change is true.

For example, you can use a state machine to represent a car's automatic transmission. The transmission has a number of operating states: park, reverse, neutral, drive, and low. As the driver shifts from one position to another the system makes a transition from one state to another, for example, from park to reverse.

Finite State Machine Representations

Traditionally, designers used truth tables to represent relationships among the inputs, outputs, and states of a finite state machine. The resulting table describes the logic necessary to control the behavior of the system under study. Another approach to designing event-driven systems is to model the behavior of the system by describing it in terms of transitions among states. The state

that is active is determined based on the occurrence of events under certain conditions. State-transition diagrams and bubble diagrams are graphical representations based on this approach.

Stateflow Representations

Stateflow uses a variant of the finite state machine notation established by Harel [1]. Using Stateflow, you create Stateflow diagrams. A Stateflow diagram is a graphical representation of a finite state machine, where *states* and *transitions* form the basic building blocks of the system. You can also represent flow (stateless) diagrams using Stateflow. Stateflow provides a block that you include in a Simulink model. The collection of Stateflow blocks in a Simulink model is the Stateflow machine.

Additionally, Stateflow enables the representation of hierarchy, parallelism, and history. Hierarchy enables you to organize complex systems by defining a parent/offspring object structure. For example, you can organize states within other higher-level states. A system with parallelism can have two or more orthogonal states active at the same time. History provides the means to specify the destination state of a transition based on historical information. These characteristics enhance the usefulness of this approach and go beyond what state-transition diagrams and bubble diagrams provide.

Notation

Notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of the following:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects

See “Stateflow Notation” on page 3-1 for detailed information on Stateflow notations.

Semantics

Semantics describe how the notation is interpreted and implemented. A completed Stateflow diagram illustrates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Knowledge of the semantics is important to make sound Stateflow diagram design decisions for code generation. Different use of notations results in different ordering of simulation and generated code execution.

The default semantics provided with the product are described in Chapter 4, "Stateflow Semantics."

References

For more information on finite state machine theory, consult these sources:

[1] Harel, David, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, 1987, pages 231-274.

[2] Hatley, Derek J., and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

Stateflow and Simulink

Stateflow functions as a finite state machine within a Simulink model. The following topics examine how Stateflow functions in this environment and how Simulink and Stateflow communicate with each other:

- "The Simulink Model and the Stateflow Machine" on page 2-5 — Describes the Stateflow machine which is the collection of Stateflow blocks in a Simulink model.
- "Stateflow Data Dictionary of Objects" on page 2-6 — Describes the Stateflow data dictionary, which is the hierarchical collection of all Stateflow objects in a Simulink model.
- "Defining Stateflow Interfaces to Simulink" on page 2-7 — Tells you how each Stateflow block in Simulink interfaces with the rest of the Simulink model.
- "Stateflow Diagram Objects" on page 2-10 — Describes most of the Stateflow objects used in creating Stateflow diagrams.

The Simulink Model and the Stateflow Machine

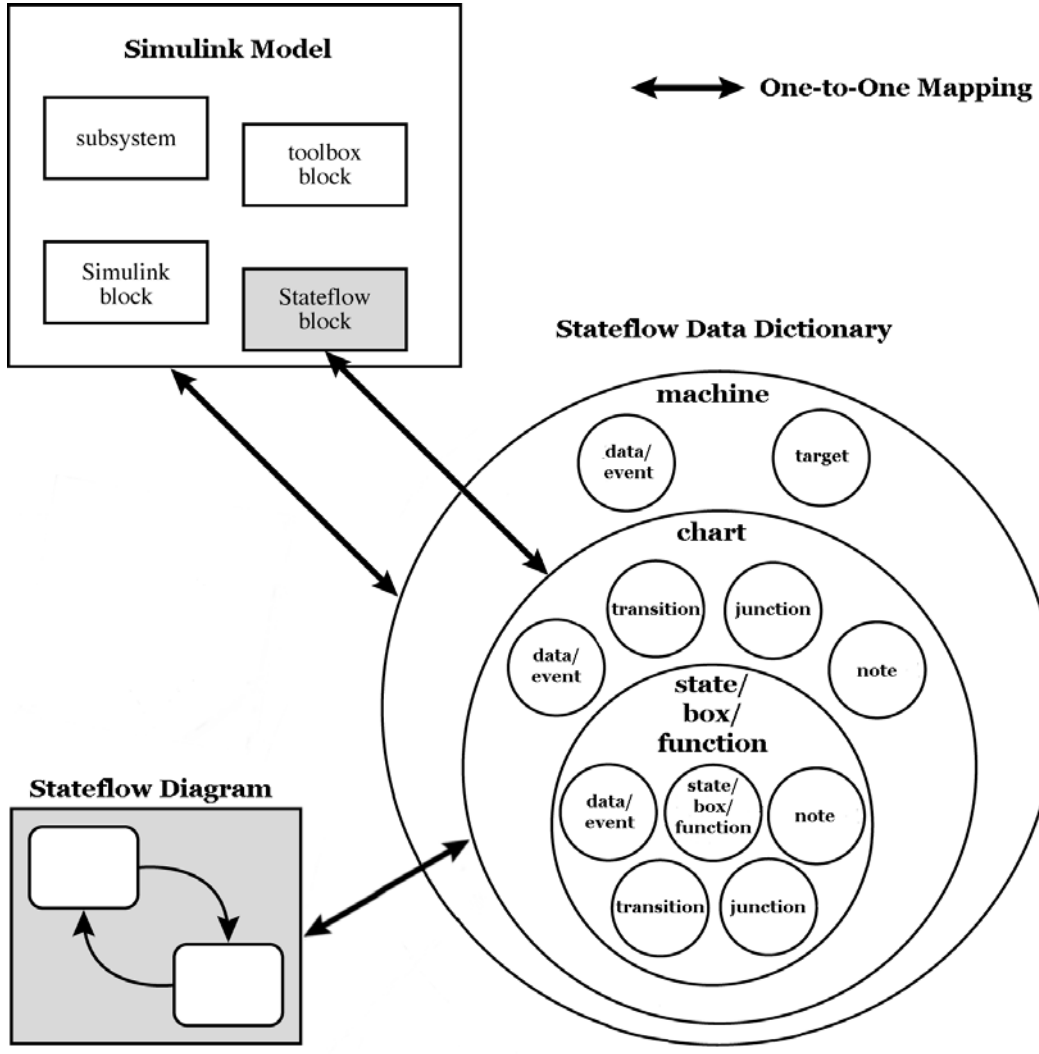
The Stateflow machine is the collection of Stateflow blocks in a Simulink model. The Simulink model and Stateflow machine work seamlessly together. Running a simulation automatically executes both the Simulink and Stateflow portions of the model.

A Simulink model can consist of combinations of Simulink blocks, toolbox blocks, and Stateflow blocks (Stateflow diagrams). In Stateflow, the chart (Stateflow diagram) consists of a set of graphical objects (states, boxes, functions, notes, transitions, connective junctions, and history junctions) and nongraphical objects (events, data, and targets).

There is a one-to-one correspondence between the Simulink model and the Stateflow machine. Each Stateflow block in the Simulink model is represented in Stateflow by a single chart (Stateflow diagram). Each Stateflow machine has its own object hierarchy. The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of graphical and nongraphical objects.

Stateflow Data Dictionary of Objects

The Stateflow data dictionary is the internal representation for the hierarchy of all Stateflow objects, graphical and nongraphical, that reside in a Simulink model. It is represented by the following diagram:



Stateflow scoping rules dictate where different nongraphical objects can exist in the hierarchy. For example, data and events can be parented by the machine, the chart (Stateflow diagram), or by a state. Targets can only be parented by the machine. Once a parent is chosen, that object is known in the hierarchy from the parent downward (including the parent's offspring). For example, a data object parented by the machine is accessible by that machine, by any charts within that machine, and by any states within that machine.

The hierarchy of the graphical objects is easily and automatically handled for you by the graphics editor. You manage the hierarchy of nongraphical objects through the Explorer or the graphics editor **Add** menu. See "Stateflow Hierarchy of Objects" on page 2-21.

Defining Stateflow Interfaces to Simulink

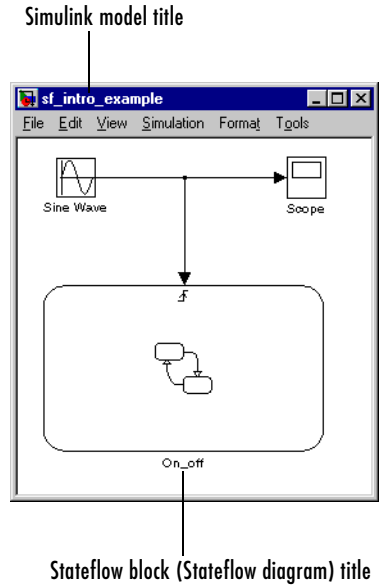
Each Stateflow block corresponds to a single Stateflow diagram. The Stateflow block interfaces to its Simulink model. The Stateflow block can interface to code sources external to the Simulink model (data, events, custom code).

Stateflow diagrams are event driven. Events can be local to the Stateflow block or can be propagated to and from Simulink and code sources external to Simulink. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to code sources external to the Simulink model.

You must define the interface to each Stateflow block. Defining the interface for a Stateflow block can involve some or all of these tasks:

- Defining the Stateflow block update method
- Defining **Output to Simulink** events
- Adding and defining nonlocal events and nonlocal data within the Stateflow diagram
- Defining relationships with any external sources

In the following example, the Simulink model titled `sf_intro_example` consists of a Simulink Sine Wave source block, a Simulink Scope sink block, and a single Stateflow block, titled `On_off`.



See “Defining Input Events” on page 6-8 and “Defining Stateflow Interfaces” on page 8-1 for more information.

Stateflow Graphical Components

Stateflow consists of these primary graphical components:

- Stateflow graphics editor (see "Create a Stateflow Diagram" on page 1-10, and "Creating a Stateflow Chart" on page 5-3)
- Stateflow Explorer (see "The Stateflow Explorer Tool" on page 10-3)
- Stateflow Coder (code generation; see "Overview of Stateflow Targets" on page 11-3)

  Stateflow Coder generates code for simulation and nonsimulation targets. A Stateflow Coder license is required to generate code for nonsimulation targets.
- Stateflow Debugger (see "Overview of the Stateflow Debugger" on page 12-2)
- Stateflow Dynamic Checker

  Stateflow Dynamic Checker supports run-time checking for conditions such as cyclic behavior and data range violations. The Dynamic Checker is currently available if you have a Stateflow license.

Stateflow Diagram Objects

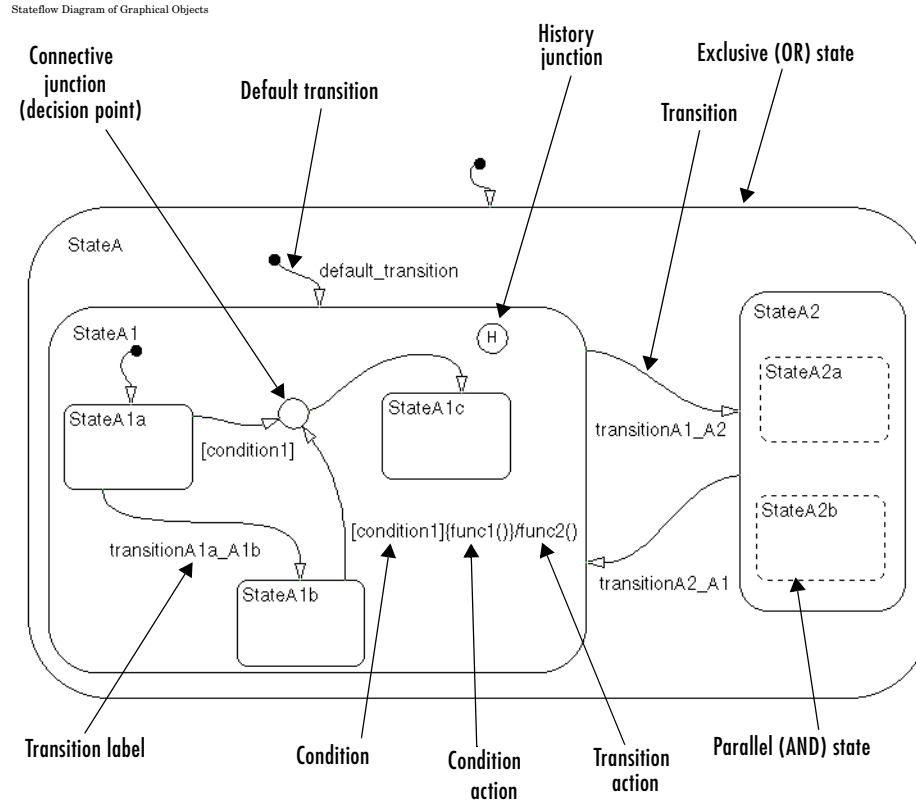
This section describes most of the graphical and nongraphical objects in a Stateflow diagram along with the concepts that relate them. The subsections are as follows:

- "Graphical Objects Example Diagram" — Presents an example display of the key graphical objects in a Stateflow diagram.
- "States" on page 2-11 — Describes the role of *states* in a Stateflow diagram that represent the current mode of an executing diagram.
- "Transitions" on page 2-13 — Describes the role of a *transitions* in a Stateflow diagram that provide a path for an executing diagram to change from one mode (state) to another.
- "Default Transitions" on page 2-14 — Describes the role of a *default transition* that specifies which exclusive (OR) state in an executing diagram is active on startup.
- "Events" on page 2-15 — Describes the role of *events* that drive Stateflow diagram execution.
- "Data" on page 2-15 — Describes the role of *data* that serve as variables in a Stateflow diagram.
- "Conditions" on page 2-16 — Describes the *conditions* that, along with events, drive Stateflow diagram execution.
- "History Junction" on page 2-17 — Describes the role of *history junctions* that record the most recently active substate of a chart or superstate.
- "Actions" on page 2-18 — Describes the role of *actions* that take place as part of Stateflow diagram execution.
- "Connective Junctions" on page 2-19 — Describes the role of *connective junctions* that provide decision points for transitions taking place during diagram execution.

All Stateflow objects are arranged in a hierarchy of objects. See "Stateflow Hierarchy of Objects" on page 2-21.

Graphical Objects Example Diagram

The following sample Stateflow diagram displays the key graphical objects of a Stateflow diagram. These objects are described in following sections that refer to this diagram.



States

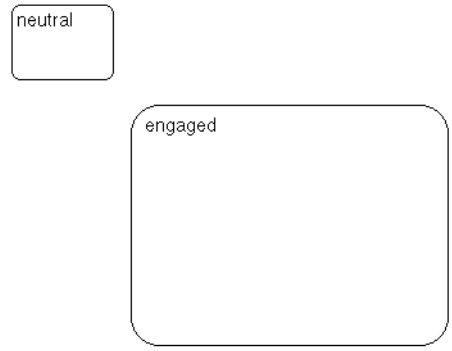
A *state* describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has a parent. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself (also called the Stateflow diagram root). You can place states within other higher-level states. In the preceding figure, `StateA1` is a child of `StateA`.

A state can have its activity history recorded in a *history junction*. History provides an efficient means of basing future activity on past activity. See "History Junction" on page 2-17.

States have labels that can specify actions executed in a sequence based upon action type. The action types are `entry`, `during`, `exit`, and `on`. See "Actions" on page 2-18.

The *decomposition* of a state defines the kind of state that a state can contain and the next level of containment. Stateflow provides two types of states: exclusive (OR) and parallel (AND) states. Exclusive (OR) states are used to describe modes that are mutually exclusive. A chart or state that contains exclusive (OR) states is said to have exclusive decomposition. The following transmission example has exclusive (OR) states.

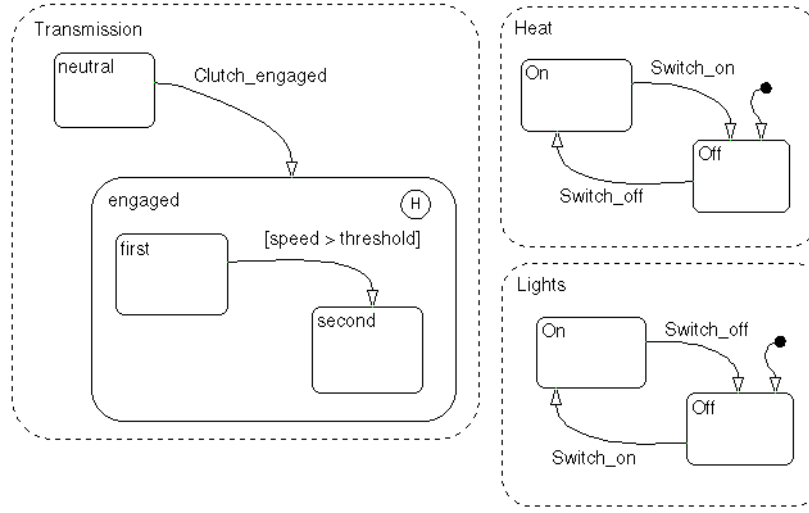


An automatic transmission can be set to either neutral or engaged. In this example either the `neutral` state or the `engaged` state is active at any one time. Both cannot be active at the same time.

A chart or state with *parallel* states has two or more states that can be active at the same time. A chart or state that contains parallel (AND) states is said to have parallel decomposition.

Parallel (AND) states are displayed as dashed rectangles. The activity of each parallel state is essentially independent of other states. In the diagram on page 2-11, `StateA2` has parallel (AND) state decomposition. Its states, `StateA2a` and `StateA2b`, are parallel (AND) states.

The following Stateflow diagram has parallel superstate decomposition.

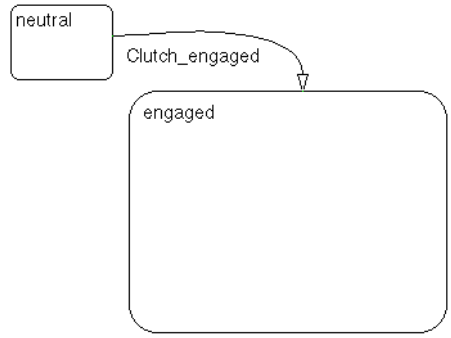


In this example, the transmission, heating, and light systems are parallel subsystems in a car. They are active at the same time and are physically independent of each other. There are many other parallel components in a car, such as the braking and windshield wiper subsystems.

Transitions

A *transition* is a graphical object that, in most cases, links one object to another. One end of a transition is attached to a source object and the other end to a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. A *transition label* describes the circumstances under which the system moves from one state to another. It is always the occurrence of some event that causes a transition to take place. In the diagram on page 2-11, the transition from `StateA1` to `StateA2` is labeled with the event `transitionA1_A2` that triggers the transition to occur.

Consider again the automatic transmission system. `clutch_engaged` is the event required to trigger the transition from `neutral` to `engaged`.

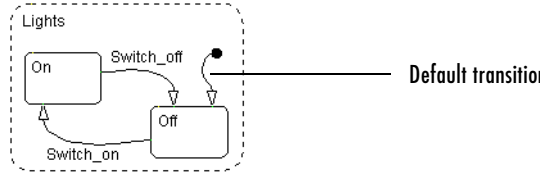


Default Transitions

Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy.

For example, in the diagram on page 2-11, the default transition to `StateA1` resolves the ambiguity that exists with regard to whether `StateA1` or `StateA2` should be active when State A becomes active. In this case, when `StateA` is active, by default `StateA1` is also active.

In the following `Lights` subsystem, the default transition to the `Lights.Off` substate indicates that when the `Lights` superstate becomes active, the `Off` substate becomes active by default.



Note History junctions override default transition paths in superstates with exclusive (OR) decomposition.

Note In parallel (AND) states, a default transition must always be present to indicate which of its exclusive (OR) states is active when the parallel state becomes active.

Events

Events drive the Stateflow diagram execution but are nongraphical objects and are thus not represented directly in a Stateflow chart. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are created and modified using the Stateflow Explorer. Events can be created at any level in the hierarchy. Events have properties such as a `scope`. The `scope` defines whether the event is

- Local to the Stateflow diagram
- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Data

Data objects are used to store numerical values for reference in the Stateflow diagram. They are nongraphical objects and are thus not represented directly in a Stateflow chart.

You create and modify data objects for Stateflow diagrams in Stateflow Explorer. Data objects have a property called `scope` that defines whether the data object is

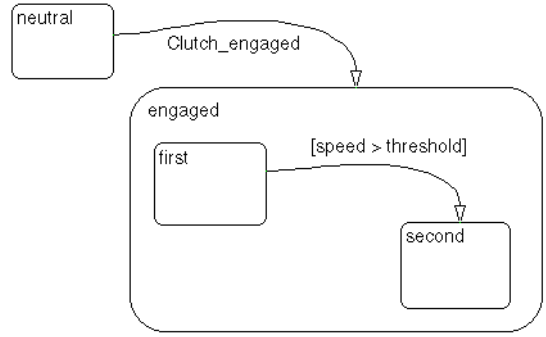
- Local to the Stateflow diagram

- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Nonpersistent temporary data
- Defined in the MATLAB workspace
- A constant
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Conditions

A *condition* is a Boolean expression specifying that a transition occurs, given that the specified expression is true. In the component summary Stateflow diagram, `[condition1]` represents a Boolean expression that must be true for the transition to occur.

In the automatic transmission system, the transition from `first` to `second` occurs if the transition condition `[speed > threshold]` is true.

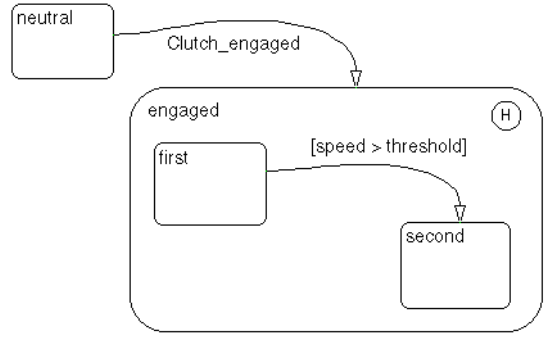


History Junction

A *history junction* records the most recently active state of a chart or superstate.

If a superstate with exclusive (OR) decomposition has a history junction, the destination substate is defined to be the substate that was most recently visited. A history junction applies to the level of the hierarchy in which it appears. The history junction overrides any default transitions. In the component summary Stateflow diagram, the history junction in `StateA1` indicates that when a transition to `StateA1` occurs, the substate that becomes active (`StateA1a`, `StateA1b`, or `StateA1c`) is based on which of those substates was most recently active.

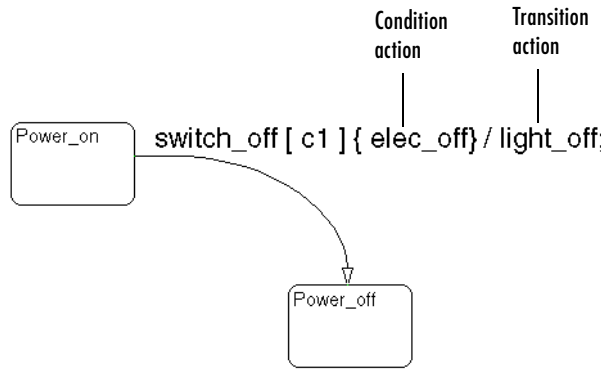
In the automatic transmission system, history indicates that when `clutch_engaged` causes a transition from `neutral` to the `engaged` superstate, the substate that becomes active, either `first` or `second`, is based on which of those substates was most recently active.



Actions

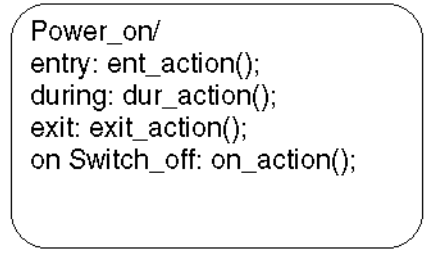
Actions take place as part of Stateflow diagram execution. The action can be executed either as part of a transition from one state to another or based on the activity status of a state.

Transitions ending in a state can have *condition* actions and *transition* actions, as shown in the following example:



In the diagram on page 2-11, the transition segment from `StateA1b` to the connective junction is labeled with the condition action `func1()` and the transition action `func2()`. The semantics of how and why actions take place are discussed throughout the examples listed in "Semantic Examples" on page 4-21.

States can have `entry`, `during`, `exit`, and `on` *`event_name`* actions. For example,



Action language defines the types of actions you can specify and their associated notations. An action can be a function call, the broadcast of an event, the assignment of a value to a variable, and so on.

Stateflow supports both Mealy and Moore finite state machine modeling paradigms. In the Mealy model, actions are associated with transitions, whereas in the Moore model they are associated with states. Stateflow supports state actions, transition actions, and condition actions. For more information, see the following:

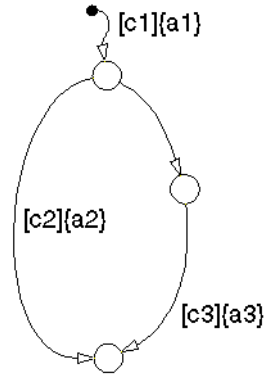
- "State Label Notation" on page 3-10
- "Transition Label Notation" on page 3-15
- "Action Language" on page 3-42
- "Action Language" on page 7-1

Connective Junctions

Connective junctions are decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior. In the diagram on page 2-11, the connective junction is used as a decision point for two transition segments that complete at `StateA1c`.

Transitions connected to junctions are called *transition segments*. Transitions, apart from default transitions, must go state to state. However, once the transition segments taken complete a state to state transition, the accumulation of the transition segments taken forms a complete transition.

The following example shows how connective junctions (displayed as small circles) are used to represent the flow of an `if-else` code structure shown in accompanying pseudocode.




```
if [c1]{
     a1
     if [c2]{
         a2
     }else if [c3]{
       a3
     }
}
```

This example executes as follows:

1. If condition `[c1]` is true, condition action `a1` is executed and the default transition to the top junction is taken.

2. Stateflow now considers which transition segment to take out of the top junction (it can take only one). Junctions with conditions have priority over junctions without conditions, so the transition with the condition `[c2]` is considered first.

3. If condition `[c2]` is true, action `a2` is executed and the transition segment to the bottom junction is taken. Because there are no outgoing transition segments from the bottom junction, the diagram is finished executing.

4. If condition `[c2]` is false, the empty transition segment on the right is taken (because it has no condition at all).

5. If condition `[c3]` is true, condition action `a3` is executed and the transition segment from the middle to the bottom junction is taken. Because there are no outgoing transition segments from the bottom junction, the diagram is finished executing.

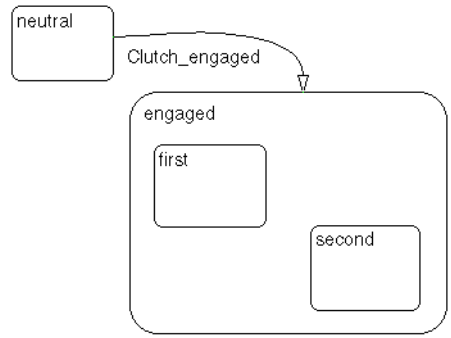
6. If condition `[c3]` is false, execution is finished at the middle junction.

The above steps describe the execution of the example diagram for connective junctions with Stateflow semantics. Stateflow semantics describe how objects in diagrams relate to each other during execution. See “Stateflow Semantics” on page 4-1.

Stateflow Hierarchy of Objects

The Stateflow hierarchy of objects enables you to organize complex Stateflow diagrams by defining a parent and child object containment structure. A hierarchical design usually reduces the number of transitions and produces neat, manageable diagrams. Stateflow supports a hierarchical organization of both charts and states. Charts can exist within charts. A chart that exists in another chart is known as a *subchart*.

Similarly, states can exist within other states. Stateflow represents state hierarchy with superstates and substates. For example, this Stateflow diagram has a superstate that contains two substates.



The `engaged` superstate contains the `first` and `second` substates. The `engaged` superstate is the parent in the hierarchy to the states `first` and `second`. When the event `clutch_engaged` occurs, the system transitions out of the `neutral` state to the `engaged` superstate. Transitions within the `engaged` superstate are intentionally omitted from this example for simplicity.

A transition out of a higher level, or *superstate*, also implies transitions out of any active substates of the superstate. Transitions can cross superstate boundaries to specify a substate destination. If a substate is made active its parent superstate is also made active.

Exploring a Real-World Stateflow Application

The modeling of a fault-tolerant fuel control system demonstrates how Simulink and Stateflow can be used to efficiently model hybrid systems containing both continuous dynamics and complex logical behavior.

Simulink elements model behavior based on a given sample time. Each loop of its block diagram is assigned an increment of sample time. Stateflow execution makes no consideration for sample time. Internally, its application might take many cycles of execution, which are assumed to take place during the sample time assigned in Simulink.

The model described represents a fuel control system for a gasoline engine. This robust control system reacts to the detection of individual sensor failures and is dynamically reconfigured for uninterrupted operation. This section describes how Stateflow is used to implement supervisory logic control system to deal with the sensor failures and contains the following topics:

- "Overview of the "fuel rate controller" Model" on page 2-22 — Gives you a top-down look at the fuel rate controller demo model along with an introduction to the Simulink logic of the fuel rate controller model and how failures are simulated.
- "Control Logic of the "fuel rate controller" Model" on page 2-25 — Introduces you to the control logic Stateflow diagram of the fuel rate controller model and its response mechanisms for system failure.
- "Simulating the "fuel rate controller" Model" on page 2-28 — Takes you step by step through the simulation of the fuel rate controller model focusing on Stateflow simulation in which you can witness state changes in response to simulated failures.

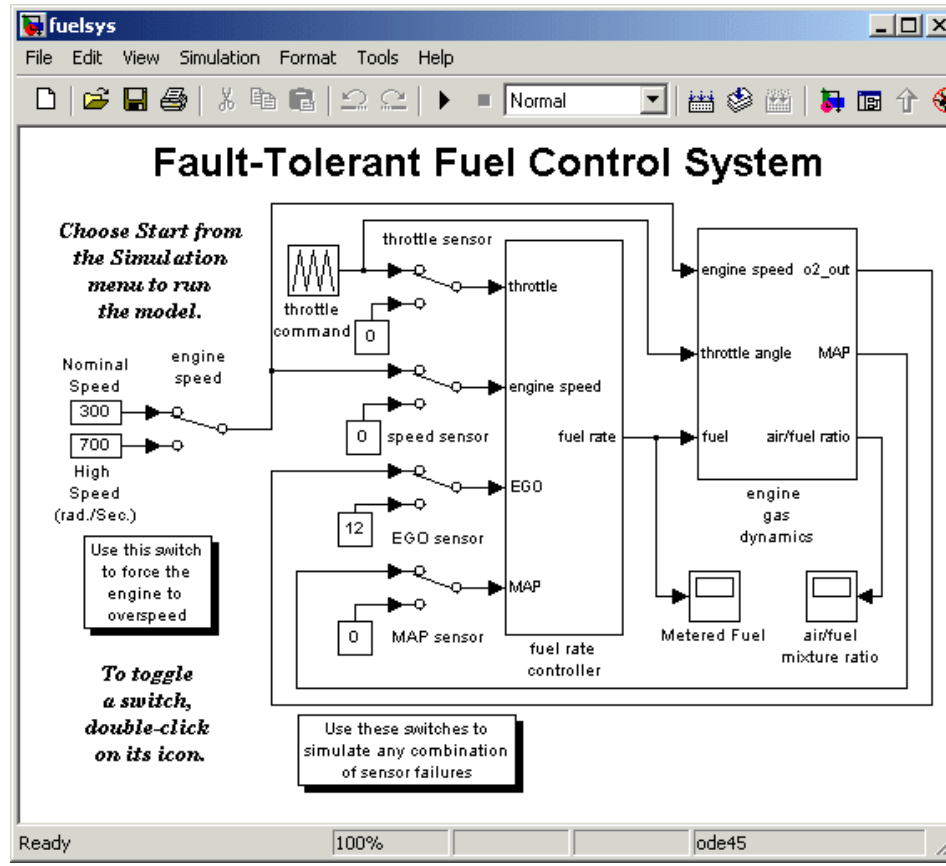
Overview of the "fuel rate controller" Model

The mass flow rate of air pumped from the intake manifold, divided by the fuel rate, which is injected at the valves, gives the air/fuel ratio. The ideal mixture ratio provides a good compromise between power, fuel economy, and emissions. A target air/fuel ratio of 14.6 is assumed in this system.

A sensor (EGO) determines the amount of residual oxygen present in the exhaust gas. This gives a good indication of the air/fuel ratio and provides a feedback measurement for closed-loop control. If the sensor indicates a high oxygen level, the controller increases the fuel rate. If the sensor detects a

fuel-rich mixture (corresponding to a very low level of residual oxygen), the controller decreases the fuel rate.

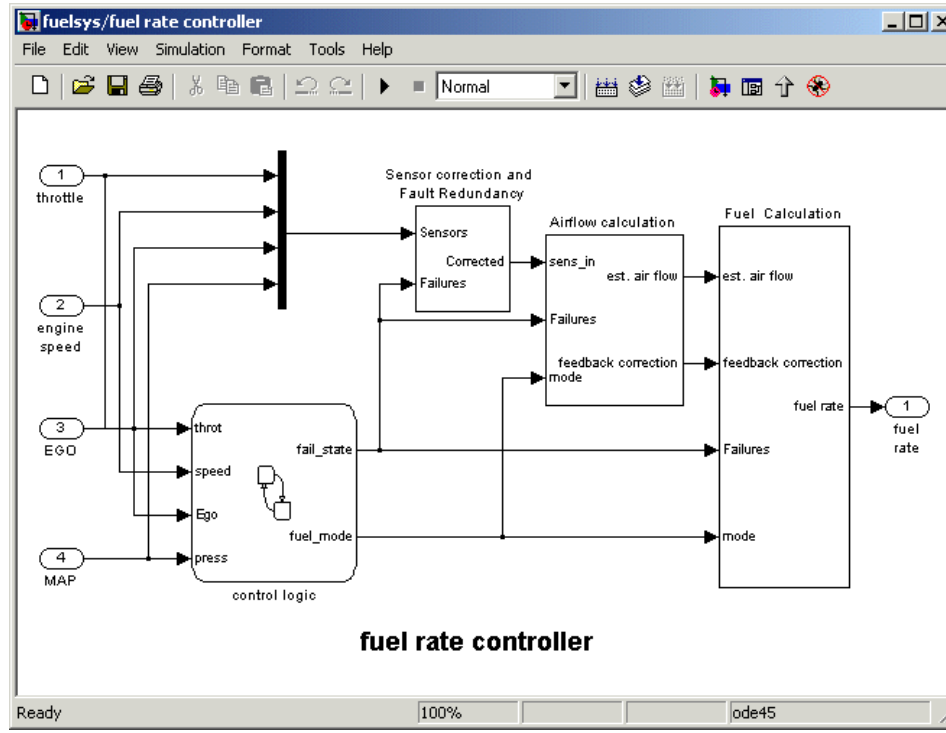
The following figure shows the top level of the Simulink model (`fuelsys.mdl`). The model is modularized into a fuel rate controller and a subsystem to simulate engine gas dynamics.



The fuel rate controller uses signals from the system's sensors to determine the fuel rate that gives an ideal mixture. The fuel rate combines with the actual air flow in the engine gas dynamics model to determine the resulting mixture ratio as sensed at the exhaust.

To simulate failures in the system, the user can selectively disable each of the four sensors: throttle angle, speed, exhaust gas (EGO), and manifold absolute pressure (MAP). Simulink accomplishes this with Manual Switch blocks. The user can toggle the position of a switch by double-clicking its icon prior to or during a simulation. Similarly, the user can induce the failure condition of a high engine speed by toggling the switch on the far left.

The controller uses the sensor input and feedback signals to adjust the fuel rate to provide an ideal ratio. The model uses four subsystems to implement this strategy: control logic, sensor correction, airflow calculation, and fuel calculation. Under normal operation, the model estimates the airflow rate and multiplies the estimate by the reciprocal of the desired ratio to give the fuel rate. Feedback from the oxygen sensor provides a closed-loop adjustment of the rate estimation in order to maintain the ideal mixture ratio.



A detailed explanation of the Simulink part of the fault-tolerant control system is given in *Using Simulink and Stateflow in Automotive Applications*, a Simulink-Stateflow Technical Examples booklet published by The MathWorks. This section concentrates on the supervisory logic part of the system that is implemented in Stateflow, but the following points are crucial to the interaction between Simulink and Stateflow:

- The supervisory logic monitors the input data readings from the sensors.
- The logic determines from these readings the sensors that have failed and outputs a failure state Boolean array as `fail_state`.
- Given the current failure state, the logic determines in which fueling mode the engine should be run.

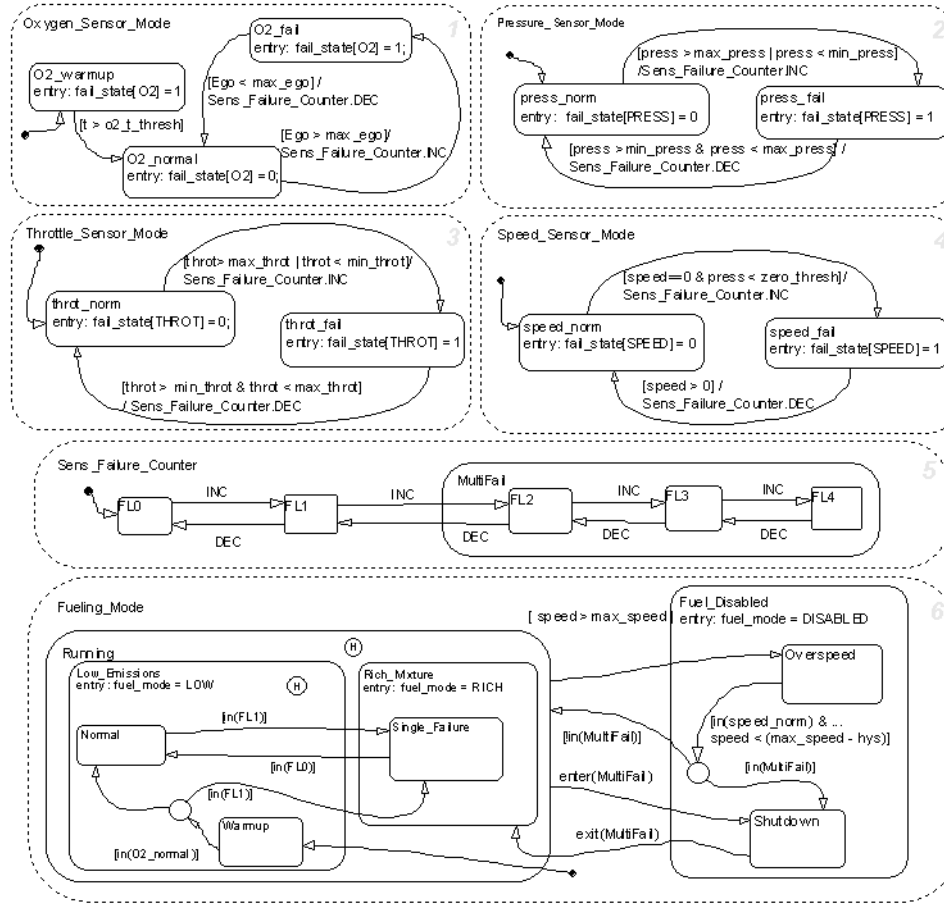
The fueling mode can be one of the following modes:

- **Low emissions mode** is the normal mode of operation where no sensors have failed.
- **Rich mixture mode** occurs when a sensor has failed, to ensure smooth running of the engine.
- **Shutdown mode** occurs when more than one sensor has failed, rendering the engine inoperable.

The fueling mode and failure state are output from Stateflow as `fuel_mode` and `fail_state` respectively into the algorithmic part of the model, where they determine the fueling calculations.

Control Logic of the "fuel rate controller" Model

The single Stateflow chart that implements the entire control logic for the `fuelsys` model is shown in the following diagram:



The chart consists of six parallel states with dashed boundaries that represent concurrent modes of operation.

The four parallel states at the top of the diagram correspond to the four individual sensors. Each of these states has a substate that represents the functioning or failing status of that sensor. These substates are mutually exclusive. For example, if the throttle sensor fails then the lone active substate of the `Throttle_Sensor_Mode` state is `throt_fail`.

Transitions determine how states can change and can be guarded by conditions. For example, the active state can change from the `throt_norm` state

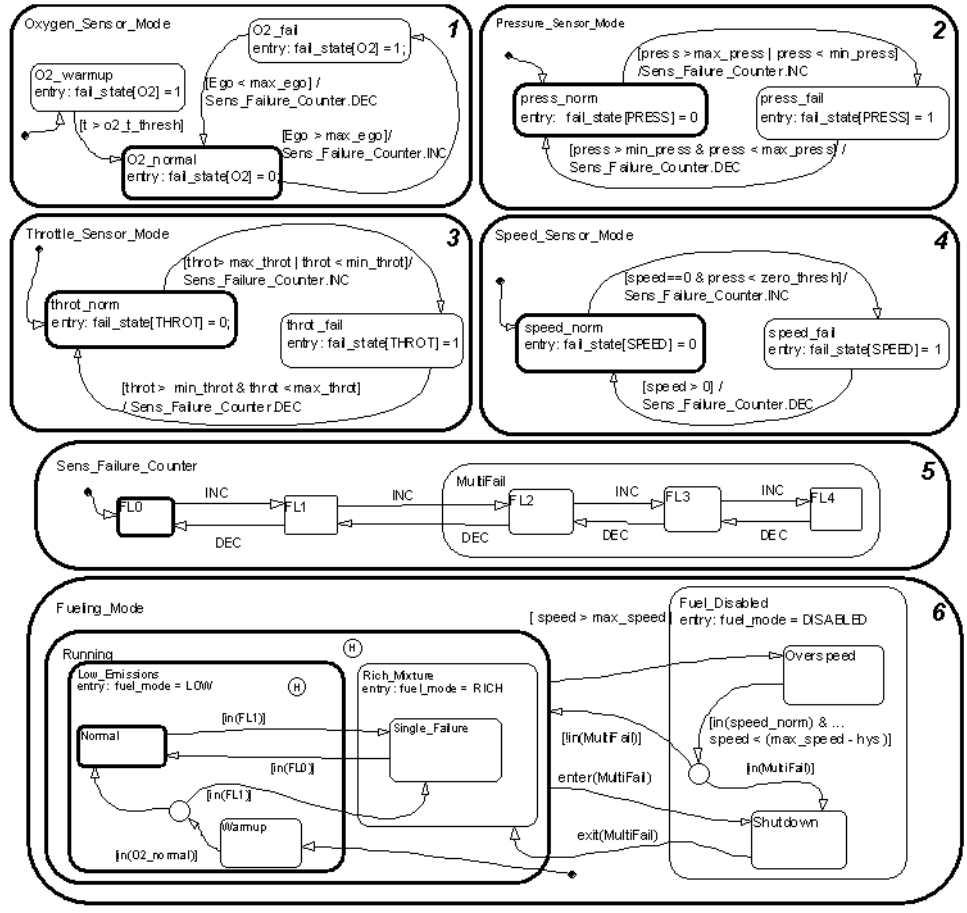
to the `throt_fail` state when the measurement from the throttle sensor exceeds `max_throt` or is below `min_throt`.

The remaining two parallel states at the bottom consider the status of the four sensors simultaneously and determine the overall system operating mode. The `Sens_Failure_Counter` superstate acts as a store for the resultant number of sensor failures. This state is polled by the `Fueling_Mode` state that determines the fueling mode of the engine. If a single sensor fails, operation continues but the air/fuel mixture is richer to allow smoother running at the cost of higher emissions. If more than one sensor has failed, the engine shuts down as a safety measure, because the air/fuel ratio cannot be controlled reliably.

Although it is possible to run Stateflow charts asynchronously by injecting events from Simulink when required, the fueling control logic is polled synchronously at a rate of 100 Hz. Consequently, the sensors are checked every 1/100 second to see if they have changed status, and the fueling mode is adjusted accordingly.

Simulating the "fuel rate controller" Model

On starting the simulation, and assuming no sensors have failed, the Stateflow diagram initializes in the `Warmup` mode in which the oxygen sensor is deemed to be in a warmup phase. If Stateflow is placed into animation mode, the current state of the system can clearly be seen highlighted on the Stateflow diagram, as shown.



After a given time period, defined by `o2_t_thresh`, the sensor is deemed to have reached operating temperature and the system settles into the normal mode of operation, shown above, in which the fueling mode is set to `NORMAL`.

As the simulation progresses, the chart is woken up synchronously every 0.01 second. The events and conditions that guard the transitions are evaluated and if a transition is valid, it is taken and animated on the Stateflow diagram.

To illustrate this, you can provoke a transition by switching one of the sensors to a failure value on the top-level Simulink model. The system detects throttle and pressure sensor failures when their measured values fall outside their nominal ranges. A manifold vacuum in the absence of a speed signal indicates a speed sensor failure. The oxygen sensor also has a nominal range for failure conditions but, because zero is both the minimum signal level and the bottom of the range, failure can be detected only when it exceeds the upper limit.

Switch the Simulink switch for the manifold air pressure sensor to the off position to witness the following sequence of transitions (note the diagram that follows).

1. Switching the Simulink manifold air pressure sensor switch causes a value of zero to be read by the fuel rate controller.

2. When the chart is next woken up, the transition from the `press_norm` state becomes valid as the reading is now out of bounds and the transition is taken to the `press_fail` state, as shown.

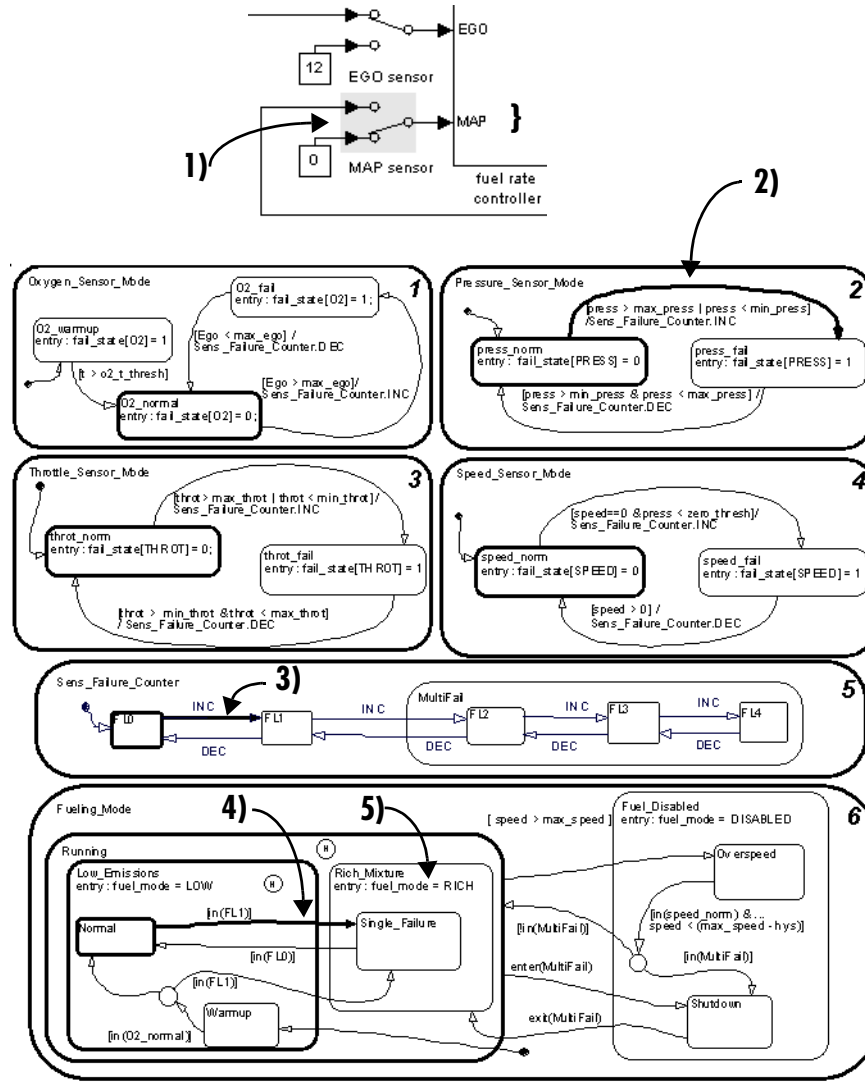
3. Regardless of which sensor fails, the model always generates the directed event broadcast `Sens_Failure_Counter.INC`, which makes the triggering of the universal sensor failure logic independent of the sensor.

   This event causes a second transition from `FL0` to `FL1` in the `Sens_Failure_Counter` superstate. Both transitions are animated on the Stateflow diagram.

4. With the `Sens_Failure_Counter` state showing one failure, the condition that guards the transition from the `Low_Emissions.Normal` state to the `Rich_Mixture.Single_Failure` state is now valid and is therefore taken.

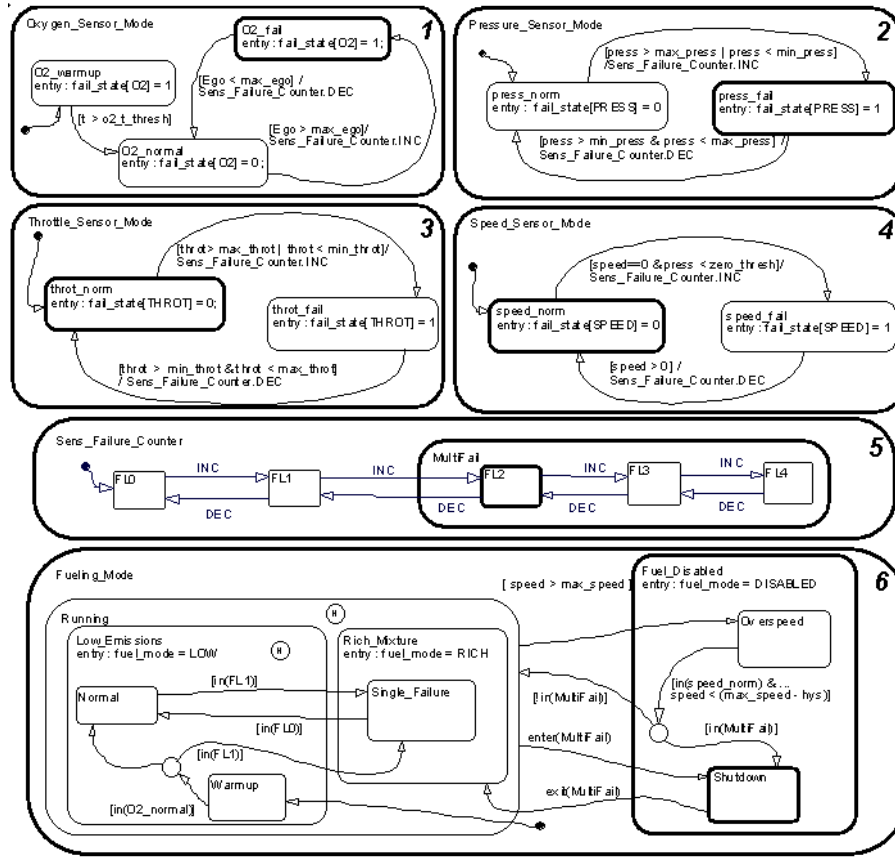
5. As the `Fuel_Disabled` state is entered, the data `fuel_mode` is set to `RICH`, as shown.

The transitions taken in the preceding steps are depicted in the following simulation diagram. Step numbers appear next to the dashed indicator line.



A second sensor failure causes the `Sens_Failure_Counter` to enter the `Multifail` state, broadcasting an implicit event that immediately triggers the

transition from the `Running` state to the `Shutdown` state. On entering the `Fuel_Disabled` superstate the Stateflow data `fuel_mode` is set to `DISABLED`.

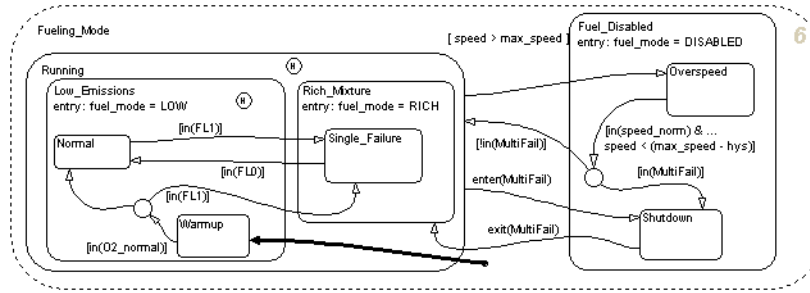


Implicit Event Broadcasts

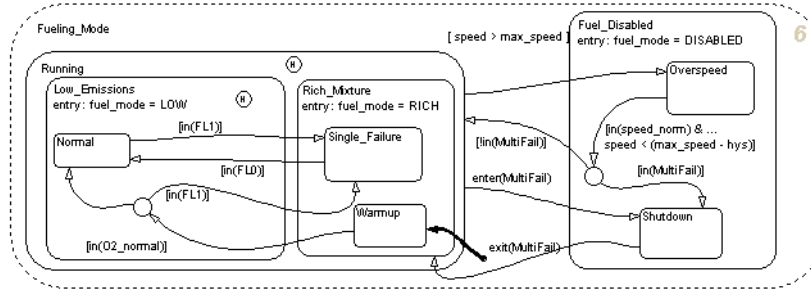
The preceding example shows how the control logic can be represented in a clear and intuitive manner. The Stateflow diagram (or chart) has been developed in a way that allows the user, or a reviewer, to easily understand how the logic is structured. Implicit event broadcasts (such as `enter(multifail)`) and implicit conditions (`in(FL0)`) make the diagram easy to read and the generated code more efficient.

Modifying the Model

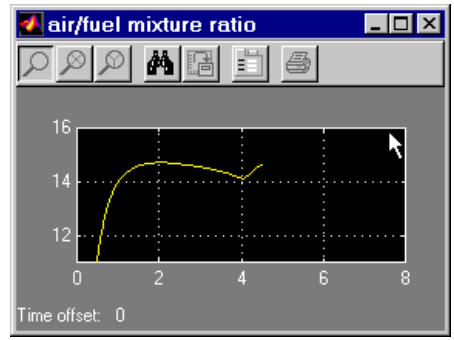
To illustrate how easy it is to modify the model, consider the `Warmup` fueling state in the fuel control logic. At the moment the fueling is set to the low emissions mode (note the highlighted default transition at the bottom).

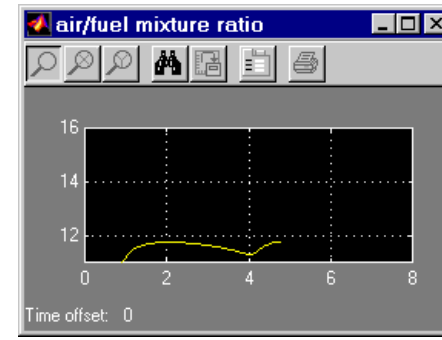


You might decide that when the oxygen sensor is warming up, changing the warmup fueling mode to a rich mixture would be beneficial. In the Stateflow chart you can easily achieve this by changing the parent of the `Warmup` state to the `Rich_Mixture` state. This is accomplished by enlarging the `Rich_Mixture` state and moving the `Warmup` state into it from the `Low_Emissions` state. This alteration is obvious to all who need to inspect or maintain the code as shown in the following result (note the highlighted default transition at the bottom):



The results of changing the algorithm can be seen in the following graphs of air/fuel mixture ratio for the first few seconds of engine operation after startup. The left graph shows the air/fuel ratio for the unaltered system. The right graph for the altered system shows how the air/fuel ratio stays low in the warming up phase indicating a rich mixture.


air/fuel mixture ratio
16
14
12
0
2
4
6
8
Time offset: 0


air/fuel mixture ratio
16
14
12
0
2
4
6
8
Time offset: 0

3

Stateflow Notation

You compose Stateflow diagrams with the symbolic objects of Stateflow notation. Learning Stateflow notation is the first step to designing and implementing efficient Stateflow diagrams. Once you are familiar with the notation of Stateflow, learn how Stateflow diagram objects interact with each other in “Stateflow Semantics” on page 4-1.

For further information on the mechanics of creating Stateflow diagrams, see “Working with Charts” on page 5-1 and “Defining Events and Data” on page 6-1.

Overview of Stateflow Objects (p. 3-3)	Stateflow contains objects that are graphical and nongraphical that are organized into an hierarchical structure.
States (p. 3-8)	States are the primary objects of Stateflow. They represent modes of a system.
Transitions (p. 3-14)	A transition is a pathway for a chart or state to change from one mode (state) to another.
Transition Connections (p. 3-18)	Stateflow supports a wide variety of connections with other Stateflow objects.
Default Transitions (p. 3-26)	Default transitions tell Stateflow which of several possible states to enter first for a chart or superstate.
Connective Junctions (p. 3-31)	A connective junction represents a decision point between alternative transition paths.
History Junctions (p. 3-38)	A history junction records the most recently active state of the chart or superstate in which it is placed.
Boxes (p. 3-40)	You use boxes to group parts of a diagram.

Graphical Functions (p. 3-41)	For convenience, Stateflow provides functions that are graphically defined by a flow graph.
Action Language (p. 3-42)	Action language defines categories of actions you include in state and transition labels.

Overview of Stateflow Objects

This section describes the different types of available Stateflow objects. Its topics are as follows:

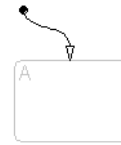
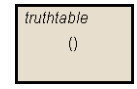
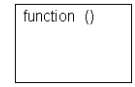
- "Graphical Objects" on page 3-3 — Introduces you to Stateflow's graphical objects (charts, states, boxes, functions, transitions, and junctions) that you draw in the Stateflow diagram editor.
- "Nongraphical Objects" on page 3-4 — Introduces you to Stateflow's nongraphical objects (data and events) that are represented textually in the Stateflow diagram editor or in the Stateflow Explorer tool.
- "The Data Dictionary" on page 3-5 — Introduces you to the data dictionary that unites all Stateflow objects (graphical and nongraphical) in a hierarchical database.
- "How Hierarchy Is Represented" on page 3-5 — Shows you how Stateflow's hierarchy of objects in the data dictionary is represented for graphical and nongraphical objects.

While this chapter defines most of these relationships, they are dealt with in more detail in Chapter 4, "Stateflow Semantics," which describes the behavior of Stateflow charts.

Graphical Objects

The following table gives the name of each graphical object in Stateflow, its appearance when drawn in the diagram editor (Notation), and the toolbar icon used in drawing the object:

Name	Notation	Toolbar Icon
State		
Transition		NA

Name	Notation	Toolbar Icon
Default transition		
Connective junction		
History junction		
Truth Table function		
Box		
Graphical function		

Nongraphical Objects

Stateflow defines the nongraphical objects described in the following sections:

- “Event Objects” on page 3-4
- “Data Objects” on page 3-5
- “Target Objects” on page 3-5

Event, data, and target objects do not have graphical representations in the Stateflow diagram editor. However, you can see them in the Stateflow Explorer. See “The Stateflow Explorer Tool” on page 10-3.

Event Objects

An event is a Stateflow object that can trigger a whole Stateflow chart or individual actions in a chart. Because Stateflow charts execute by reacting to

events, you specify and program events into your charts to control their execution. You can broadcast events to every object in the scope of the object sending the event, or you can send an event to a specific object. You can define explicit events that you specify directly, or you can define implicit events to take place when certain actions are performed, such as entering a state. For a full description of events, see "Defining Events" on page 6-2.

Data Objects

A Stateflow chart stores and retrieves data that it uses to control its execution. Stateflow data resides in its own workspace, but you can also access data that resides externally in the Simulink model or application that embeds the Stateflow machine. When creating a Stateflow model, you must define any internal or external data that you use in the action language of a Stateflow chart. For a full description of data objects, see "Defining Data" on page 6-14.

Target Objects

You build targets in Stateflow to execute the application you program in Stateflow charts and the Simulink model that contains them. A target is a program that executes a Stateflow model or a Simulink model containing a Stateflow machine. You build a simulation target (named `sfun`) to execute a simulation of your model. You build a Real-Time Workshop target (named `rtw`) to execute the Simulink model on a supported processor environment. You build custom targets (with names other than `sfun` or `rtw`) to pinpoint your application to a specific environment. For a full description of target objects in Stateflow and Simulink, see "Overview of Stateflow Targets" on page 11-3.

The Data Dictionary

The data dictionary is a database containing all the information about the graphical and nongraphical objects. Data dictionary entries for graphical objects are created automatically as the objects are added and labeled. You explicitly define nongraphical objects in the data dictionary by using the Explorer. The parser evaluates entries and relationships between entries in the data dictionary to verify that the notation is correct.

How Hierarchy Is Represented

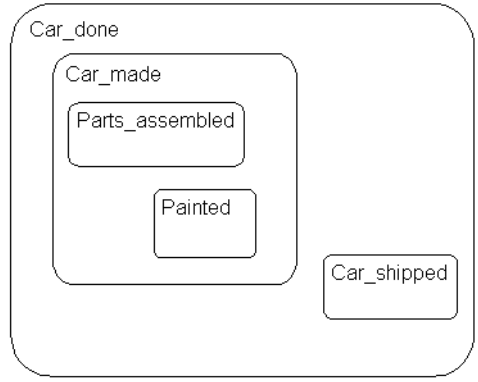
Stateflow notation supports the representation of graphical object hierarchy in Stateflow diagrams. See the following examples of representing hierarchy in Stateflow diagrams:

- "Representing State Hierarchy Example" on page 3-6
- "Representing Transition Hierarchy Example" on page 3-7

Data and event objects are nongraphical objects whose hierarchy is represented in the Explorer tool. Data and event hierarchy is defined by specifying the parent object when you create it. See Chapter 6, "Defining Events and Data" and Chapter 8, "Defining Stateflow Interfaces" for information and examples on representing data and event objects in the Explorer tool.

Representing State Hierarchy Example

In the following example, drawing one state within the boundaries of another state indicates that the inner state is a substate or child of the outer state or superstate and the outer state is the parent of the inner state:



In this example, the Stateflow diagram is the parent of the state `Car_done`. The state `Car_done` is the parent state of the `Car_made` and `Car_shipped` states. The state `Car_made` is also the parent of the `Parts_assembled` and `Painted` states. You can also say that the states `Parts_assembled` and `Painted` are children of the `Car_made` state.

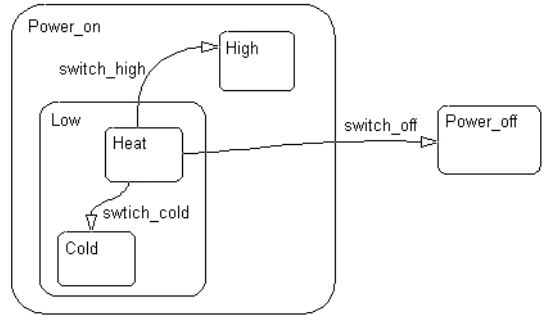
Stateflow hierarchy can also be represented textually, in which the Stateflow diagram is represented by the slash (`/`) character and each level in the hierarchy of states is separated by the period (`.`) character. The following is a textual representation of the hierarchy of objects in the preceding example:

- `/Car_done`

- `/Car_done.Car_made`
- `/Car_done.Car_shipped`
- `/Car_done.Car_made.Parts_assembled`
- `/Car_done.Car_made.Painted`

Representing Transition Hierarchy Example

This is an example of how transition hierarchy is represented.



A transition's hierarchy is described in terms of the transition's parent, source, and destination. The parent is the lowest level that contains the source and destination of the transition. The machine is the root of the hierarchy. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is separated by the period (`.`) character. The three transitions in the example are represented in the following table.

Transition Label	Transition Parent	Transition Source	Transition Destination
`switch_off`	`/`	`/Power_on.Low.Heat`	`/Power_off`
`switch_high`	`/Power_on`	`/Power_on.Low.Heat`	`/Power_on.High`
`switch_cold`	`/Power_on.Low`	`/Power_on.Low.Heat`	`/Power_on.Low.Cold`

States

This section describes Stateflow's primary object, the state. States represent modes of a reactive system. See the following topics for information about states and their properties:

- "What Is a State?" on page 3-8 — Describes states as modes of behavior that can be active or inactive.
- "State Decomposition" on page 3-9 — Shows you how states can be exclusive of each other or parallel with each other in behavior in an executing diagram.
- "State Label Notation" on page 3-10 — Shows you how to specify the name of a state and its actions through its label.

What Is a State?

A *state* describes a mode of a reactive Stateflow chart. States in a Stateflow chart represent these modes. The following table shows the button icon for a drawing a state in the Stateflow diagram editor and a short description.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

State Hierarchy

States can exist as superstates, substates, and just states. A state is a superstate if it contains other states, called substates. A state is a substate if it exists in another state. A state that is neither a superstate nor a substate of another state is just a state.

Every state is part of a hierarchy. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself. A state also has history that applies to its level of hierarchy in the Stateflow diagram. States can have actions that are executed in a sequence based upon the types of its actions. The action types are `entry`, `during`, `exit`, or `on` *`event_name`*.

Active and Inactive States

When a state is active, the chart takes on that mode. When a state is inactive, the chart is not in that mode. The activity or inactivity of a chart's states dynamically changes based on events and conditions. The occurrence of events drives the execution of the Stateflow diagram by making states become active or inactive. At any point in the execution of a Stateflow diagram, there is a combination of active and inactive states.

State Decomposition

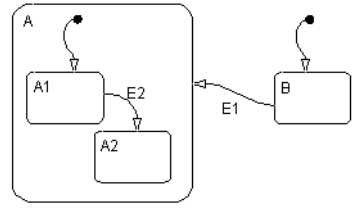
Every state (and chart) has a *decomposition* that dictates what kind of substates it can contain. All substates of a superstate must be of the same type as the superstate's decomposition. Decomposition for a state can be exclusive (OR) or parallel (AND). These types of decomposition are described in the following topics:

- "Exclusive (OR) State Decomposition" on page 3-9
- "Parallel (AND) State Decomposition" on page 3-10

Exclusive (OR) State Decomposition

Exclusive (OR) state decomposition for a superstate (or chart) is indicated when its substates have solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time. The children of exclusive (OR) decomposition parents are OR states.

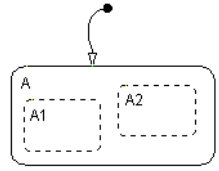
In the following example, either state `A` or state `B` can be active. If state `A` is active, either state `A1` or state `A2` can be active at any one time.



Parallel (AND) State Decomposition

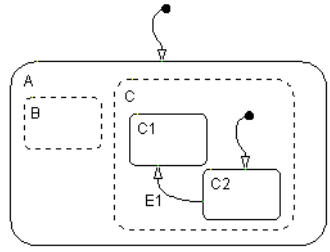
The children of parallel (AND) decomposition parents are parallel (AND) states. Parallel (AND) state decomposition for a superstate (or chart) is indicated when its substates have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are always active at the same time.

In the following example, when state `A` is active, `A1` and `A2` are both active at the same time:



The activity within parallel states is essentially independent, as demonstrated in the following example.

In the following example, when state `A` becomes active, both states `B` and `C` become active at the same time. When state `C` becomes active, either state `C1` or state `C2` can be active.



State Label Notation

The label for a state appears on the top left corner of the state rectangle with the following general format:

```
name/
entry:entry action
```

```
during:during action
exit:exit action
on event_name:on event_name action
```

A state label starts with the name of the state followed by an optional / character (see “State Name” on page 3-12) and additional lines with keyword prefixes. Each keyword prefix identifies a different type of action associated with the state. These actions are as follows (short form is shown in bold):

Prefix	Action Type
entry	Entry action. See “Entry Action” on page 3-13.
during	During action. See “During Action” on page 3-13.
exit	Exit action. See “Exit Action” on page 3-13.
on *event_name*	Action occurring with event *event_name*. See “On Event_Name Action” on page 3-13.

You enter action statements for the state after the keyword label that identifies its type of action. Separate multiple actions for each action type with one of the following characters:

- Carriage return
- Semicolon
- Comma

You can specify multiple `on` *`event_name`* actions for different events by adding multiple `on` *`event_name`* lines specifying unique values for *`event_name`*.

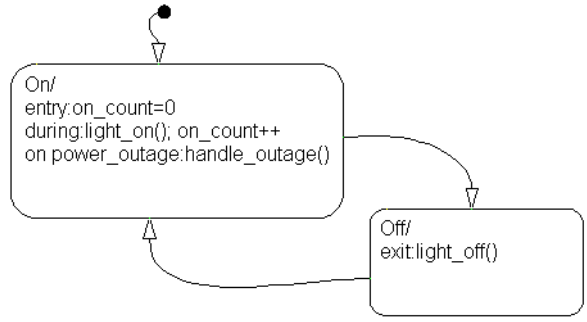
Each keyword is optional and positionally independent. You can specify none, some, or all of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

If you enter the name and slash followed directly by actions, the actions are interpreted as `entry` action(s). This shorthand is useful if you are only specifying `entry` actions.

The action you specify in state labels forms a part of Stateflow *action language*. See “Action Language” on page 3-42.

State Label Example

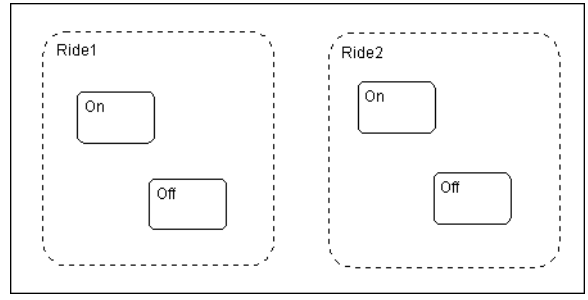
The following example demonstrates the components of a state label.



State Name. In this example, the state names are `On` and `Off`. The *name* of the state forms the first part of the state label. Valid state names consist of alphanumeric characters and can include the underscore (_) character, for example, `Transmission` or `Green_on`.

The use of hierarchy provides some flexibility in the naming of states. The name that you enter as part of the label must be unique when preceded by the hierarchy of its ancestor states. The name stored in the data dictionary is the text you enter as the label on the state, preceded by the hierarchy of its parent states separated by periods. Each state can have the same name appear in the label of the state, as long as their full names within the data dictionary are unique. Otherwise, the parser indicates an error.

The following example shows how hierarchy supports unique naming of states.



Each of these states has a unique name because of its location in the hierarchy of the Stateflow diagram. Although the name portion of the label on these states is not unique, when the hierarchy is prefixed to the name in the data dictionary, the result is unique. The full names for these states as seen in the data dictionary are as follows:

- `Ride1.On`
- `Ride1.Off`
- `Ride2.On`
- `Ride2.Off`

The following actions apply to the preceding "State Label Example" on page 3-12:

Entry Action. State `On` has `entry` action `on_count=0`. The value of `on_count` is reset to 0 whenever state `On`'s `entry` action is executed.

During Action. State `On` has two `during` actions, `light_on()` and `on_count++`. These actions are executed whenever state `On`'s `during` action is executed.

Exit Action. State `Off` has the `exit` action `light_off()`. This action is executed whenever state `Off`'s `exit` action is executed.

On Event_Name Action. State `On` has `on power_outage` action. When the event `power_outage` occurs, the action `handle_outage()` is executed.

See "Executing a State" on page 4-13 for information on how and when `entry`, `during`, `exit`, and `on` *`event_name`* actions are taken.

For more detailed information on the action language in transition labels, see Chapter 7, "Action Language."

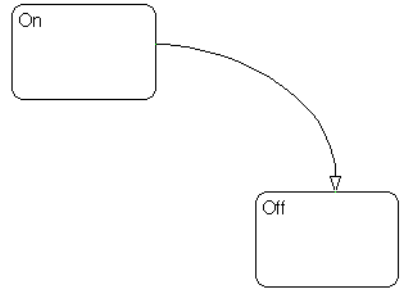
Transitions

You model the behavior of reactive systems by changing from one state to another through an object called a transition. This section contains the following topics on the transitions in Stateflow diagrams:

- "What Is a Transition?" on page 3-14 — Gives a description and examples of transitions and transition segments.
- "Transition Label Notation" on page 3-15 — Gives the syntax for a transition label, which can include conditions and actions. Also defines a condition and the different types of actions in a transition label.
- "Valid Transitions" on page 3-17 — Describes the situations under which transitions become valid and are taken during execution of the Stateflow diagram.

What Is a Transition?

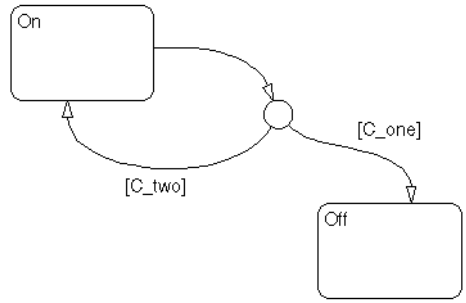
A *transition* is a curved line with an arrowhead that links one graphical object to another. In most cases, a transition represents the passage of the system from one mode (state) object to another. A transition is attached to a source and a destination object. The *source* object is where the transition begins and the *destination* object is where the transition ends. This is an example of a transition from a source state, `On`, to a destination state, `Off`.



Junctions divide a transition into transition segments. In this case, a full transition consists of the segments taken from the origin to the destination

state. Each segment is evaluated in the process of determining the validity of a full transition.

The following example has two segmented transitions: one from state On to state Off, and the other from state On to itself:



A default transition is a special type of transition that has no source object. See "Default Transitions" on page 3-26 for a description of a default transition.

Transition Label Notation

A transition is characterized by its *label*. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have the following general format:

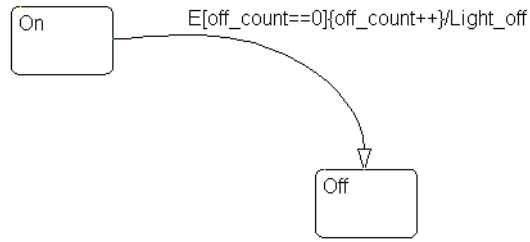
```
event[condition]{condition_action}/transition_action
```

You replace the names for `event`, `condition`, `condition_action`, and `transition_action` with appropriate contents as shown in the example "Transition Label Example" on page 3-16. Each part of the label is optional.

The action you specify in transition labels forms a part of Stateflow *action language*. See "Action Language" on page 3-42.

Transition Label Example

Use the transition label in the following example to understand the parts of a transition label.



Event. The specified *event* is what causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. Absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events are specified using the OR logical operator (`|`).

In this example, the broadcast of event `E` triggers the transition from `On` to `Off` provided the condition `[off_count==0]` is true.

Condition. A *condition* is a Boolean expression to specify that a transition occurs given that the specified expression is true. Enclose the condition in square brackets (`[]`). See "Condition Statements" on page 7-67 for information on the condition notation.

In this example, the condition `[off_count==0]` must evaluate as true for the condition action to be executed and for the transition from the source to the destination to be valid.

Condition Action. A *condition action* follows the condition for a transition and is enclosed in curly braces (`{}`). It is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. If no condition is specified, an implied condition evaluates to true and the condition action is executed. See "Action Language" on page 3-42 for more information on condition actions in action language.

In this example, if the condition `[off_count==0]` is true, the condition action `off_count++` is immediately executed.

Transition Action. The transition action is executed after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid. Precede the transition action with a backslash. See "Action Language" on page 3-42 for more information on the action language.

In this example, if the condition `[off_count==0]` is true, and the destination state `Off` is valid, the transition action `Light_off` is executed.

Valid Transitions

In most cases, a transition is valid when the source state of the transition is active and the transition label is valid. Default transitions are slightly different because there is no source state. Validity of a default transition to a substate is evaluated when there is a transition to its superstate, assuming the superstate is active. This labeling criterion applies to both default transitions and general case transitions. The following are possible combinations of valid transition labels.

Transition Label	Is Valid If...
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Condition only	Any event occurs and the condition is true
Action only	Any event occurs
Not specified	Any event occurs

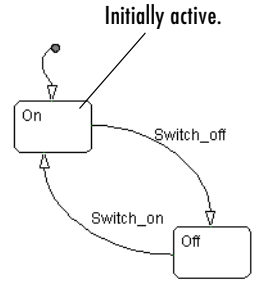
Transition Connections

Stateflow notation supports a wide variety of transition connections, which are demonstrated by the examples in the following sections:

- "Transitions to and from Exclusive (OR) States" on page 3-18 — Describes the behavior of transitions that take place between exclusive (OR) states.
- "Transitions to and from Junctions" on page 3-19 — Describes the behavior of transitions that take place between exclusive (OR) states and a connective junction.
- "Transitions to and from Exclusive (OR) Superstates" on page 3-19 — Describes the behavior of transitions that take place between a state and a superstate.
- "Transitions to and from Substates" on page 3-20 — Describes the behavior of a transition from one substate to another.
- "Self-Loop Transitions" on page 3-22 — Describes the behavior of a self-loop transition through a connective junction.
- "Inner Transitions" on page 3-22 — Describes several examples that demonstrate the behavior of inner transitions to substates and a history junction and why you want to use them.

Transitions to and from Exclusive (OR) States

This example shows simple transitions to and from exclusive (OR) states.

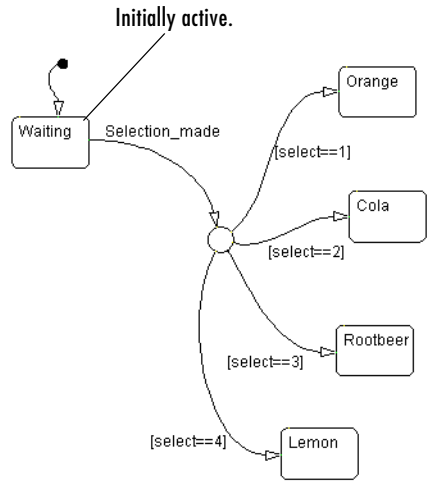


The transition `On`→`Off` is valid when state `On` is active and the event `Switch_off` occurs. The transition `Off`→`On` is valid when state `Off` is active and event `Switch_on` occurs.

See "Transitions to and from Exclusive (OR) States Examples" on page 4-23 for more information on the semantics of this notation.

Transitions to and from Junctions

This example shows transitions to and from a connective junction.

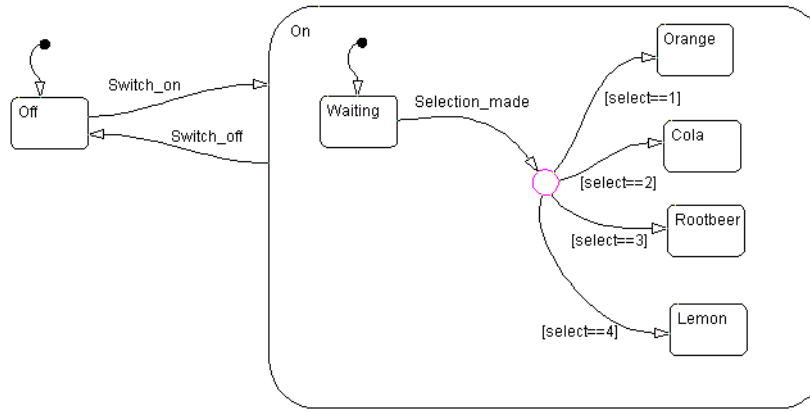


This is a Stateflow diagram of a soda machine. The Stateflow diagram is called when the external event `Selection_made` occurs. The Stateflow diagram awakens with the `Waiting` state active. The `Waiting` state is a common source state. When the event `Selection_made` occurs, the Stateflow diagram transitions from the `Waiting` state to one of the other states based on the value of the variable `select`. One transition is drawn from the `Waiting` state to the connective junction. Four additional transitions are drawn from the connective junction to the four possible destination states.

See "Transitions from a Common Source to Multiple Destinations Example" on page 4-57 for more information on the semantics of this notation.

Transitions to and from Exclusive (OR) Superstates

This example shows transitions to and from an exclusive (OR) superstate and the use of a default transition.



This is an expansion of the soda machine Stateflow diagram that includes the initial example of the `On` and `Off` exclusive (OR) states. On is now a superstate containing the `Waiting` and soda choices states. The transition `Off`→`On` is valid when state `Off` is active and event `Switch_on` occurs. Now that `On` is a superstate, this is an explicit transition to the `On` superstate.

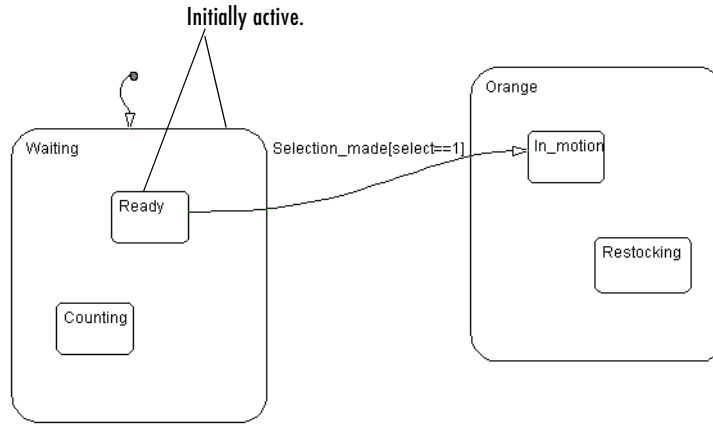
For a transition to a superstate to be a valid, the destination substate must be implicitly defined. The destination substate for `On` is implicitly defined by making the `Waiting` substate the destination state of a default transition. This notation defines that the resultant transition is made from the `Off` state to the state `On.Waiting`.

The transition from `On` to `Off` is valid when state `On` is active and event `Switch_off` occurs. However, when the `Switch_off` event occurs, a transition to the `Off` state must take place no matter which of the substates of `On` is active. This top-down approach simplifies the Stateflow diagram by looking at the transitions out of the superstate without considering all the details of states and transitions within the superstate.

See “Default Transition Examples” on page 4-35 for more information on the semantics of this notation.

Transitions to and from Substates

The following example shows transitions to and from exclusive (OR) substates.

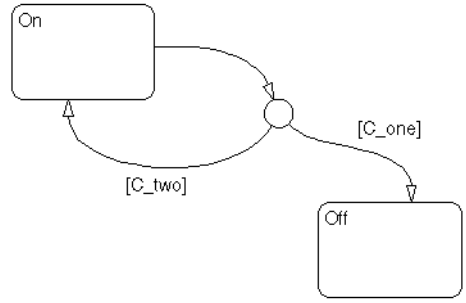


This Stateflow diagram shows a transition from one OR substate to another OR substate: the transition from `Waiting.Ready` to `Orange`. The transition to the state `In_motion` is valid when state `Waiting.Ready` is active and the event `Selection_made` occurs, providing that the variable `select` equals 1. This transition defines an explicit exit from the `Waiting.Ready` state and an implicit exit from the `Waiting` superstate. On the destination side, this transition defines an implicit `entry` into the `Orange` superstate and an explicit `entry` into the `Orange.In_motion` substate.

See “Transitioning from a Substate to a Substate with Events Example” on page 4-27 for more information on the semantics of this notation.

Self-Loop Transitions

A transition segment from a state to a connective junction that has an outgoing transition segment from the connective junction back to the state is a self-loop transition as shown in the following example:



See these sections for examples of self-loop transitions:

- "Connective Junction—Self-Loop Example" on page 3-34

  See the section "Self-Loop Transition Example" on page 4-53 for information on the semantics of this notation.
- "Connective Junction and For Loops Example" on page 3-34

  See "For Loop Construct Example" on page 4-54 for information on the semantics of this notation.

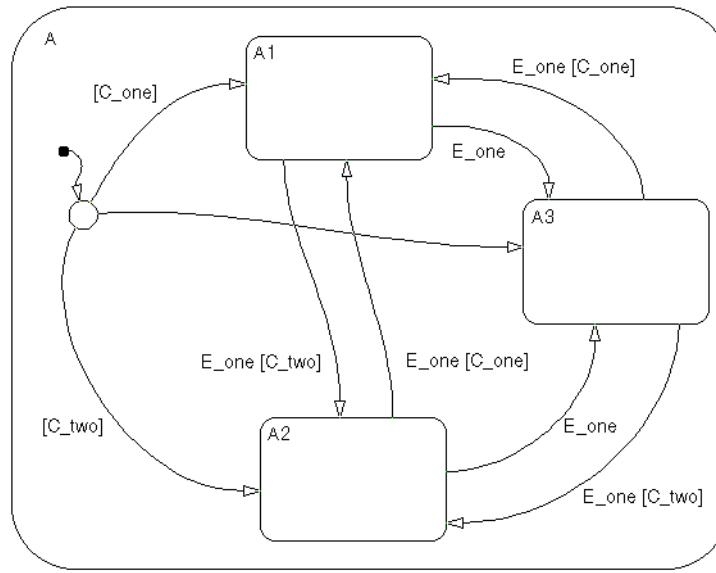
Inner Transitions

An *inner transition* is a transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify a Stateflow diagram, as shown by the following examples:

- "Before Using an Inner Transition" on page 3-23
- "After Using an Inner Transition to a Connective Junction" on page 3-24
- "Using an Inner Transition to a History Junction" on page 3-25

Before Using an Inner Transition

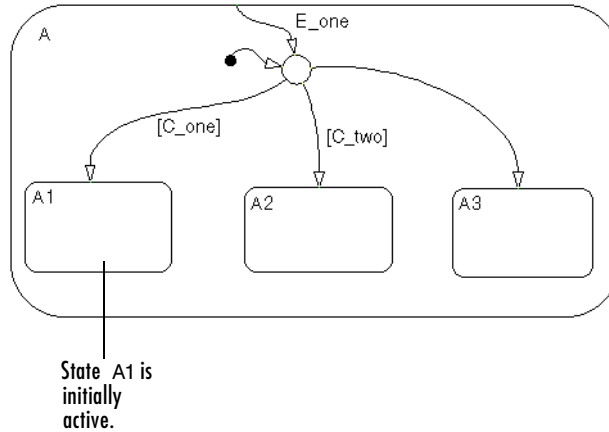
This is an example of a Stateflow diagram that could be simplified by using an inner transition.



Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transition is determined by `[C_one]` and `[C_two]`. If `[C_one]` is true, the transition to `A1` is true. If `[C_two]` is true, the transition to `A2` is valid. If neither `[C_one]` nor `[C_two]` is true, the transition to `A3` is valid. The transitions among `A1`, `A2`, and `A3` are determined by `E_one`, `[C_one]`, and `[C_two]`.

After Using an Inner Transition to a Connective Junction

This example simplifies the preceding example using an inner transition to a connective junction.



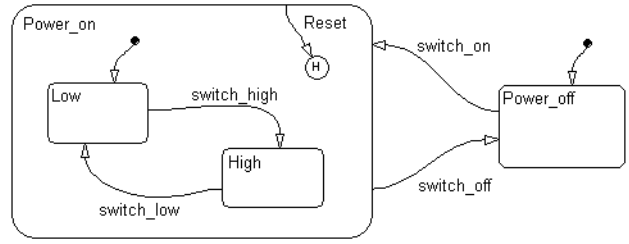
Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transitions is determined by `[C_one]` and `[C_two]`.

The Stateflow diagram is simplified by using an inner transition in place of the many transitions among all the states in the original example. If state `A` is already active, the inner transition is used to reevaluate which of the substates of state `A` is to be active. When event `E_one` occurs, the inner transition is potentially valid. If `[C_one]` is true, the transition to `A1` is valid. If `[C_two]` is true, the transition to `A2` is valid. If neither `[C_one]` nor `[C_two]` is true, the transition to `A3` is valid. This solution is much simpler than the previous one.

See “Processing the First Event with an Inner Transition to a Connective Junction” on page 4-45 for more information on the semantics of this notation.

Using an Inner Transition to a History Junction

This example shows an inner transition to a history junction.



State `Power_on.High` is initially active. When event `Reset` occurs, the inner transition to the history junction is valid. Because the inner transition is valid, the currently active state, `Power_on.High`, is exited. When the inner transition to the history junction is processed, the last active state, `Power_on.High`, becomes active (is reentered). If `Power_on.Low` was active under the same circumstances, `Power_on.Low` would be exited and reentered as a result. The inner transition in this example is equivalent to drawing an outer self-loop transition on both `Power_on.Low` and `Power_on.High`.

See “Use of History Junctions Example” on page 3-38 for another example using a history junction.

See “Inner Transition to a History Junction Example” on page 4-48 for more information on the semantics of this notation.

Default Transitions

You use default transitions to tell Stateflow which one of several states you enter when you first enter a chart or a state that has substates. See the following topics for information on default transitions:

- "What Is a Default Transition?" on page 3-26 — Defines and describes a default transition.
- "Drawing Default Transitions" on page 3-26 — Gives you the steps for drawing a default transition.
- "Labeling Default Transitions" on page 3-27 — Gives you the steps for editing the label of a default transition.
- "Default Transition Examples" on page 3-27 — Provides some examples of default transitions.

What Is a Default Transition?

Default transitions are primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. They have a destination but no source object. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information such as a history junction. Default transitions are also used to specify that a junction should be entered by default.

Drawing Default Transitions

Click the **Default transition** button in the toolbar, and click a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition. In some cases it is useful to label default transitions.

One of the most common Stateflow programming mistakes is to create multiple exclusive (OR) states without a default transition. In the absence of the default transition, there is no indication of which state becomes active by default. Note that this error is flagged when you simulate the model using the Debugger with the **State Inconsistencies** option enabled.

This table shows the button icon and briefly describes a default transition.

Name	Button Icon	Description
Default transition		Use a default transition to indicate, when entering this level in the hierarchy, which object becomes active by default.

Labeling Default Transitions

In some circumstances, you might want to label default transitions. You can label default transitions as you would other transitions. For example, you might want to specify that one state or another should become active depending upon the event that has occurred. In another situation, you might want to have specific actions take place that are dependent upon the destination of the transition.

Note When labeling default transitions, take care to ensure that there is always at least one valid default transition. Otherwise, a Stateflow chart can transition into an inconsistent state.

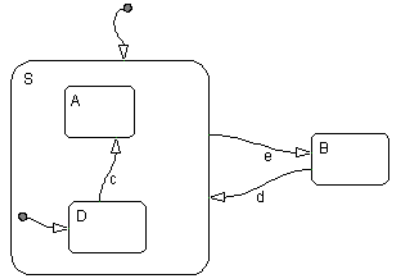
Default Transition Examples

The following examples show the use of default transitions in Stateflow diagrams:

- "Default Transition to a State Example" on page 3-28
- "Default Transition to a Junction Example" on page 3-29
- "Default Transition with a Label Example" on page 3-29

Default Transition to a State Example

This example shows a use of default transitions.



When the Stateflow diagram is first awakened, it must decide whether to activate state `S` or state `B` since they are exclusive (OR) states. The answer is given by the default transition to superstate `S`, which is taken if valid. Because there are no conditions on this default transition, it is taken.

State `S`, which is now active, has two substates, `A` and `D`. Which substate becomes active? Only one of them can be active because they are exclusive (OR) states. The answer is given by the default transition to substate `D`, which is taken if valid. Because there are no conditions on this default transition, it is taken.

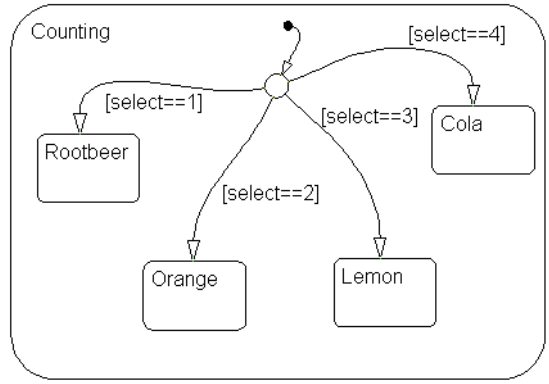
Suppose at a different execution point the Stateflow diagram is awakened by the occurrence of event `d` and state `B` is active. The transition from state `B` to state `S` is valid. When the system enters state `S`, it enters substate `D` because the default transition is defined.

See “Default Transition Examples” on page 4-35 for more information on the semantics of this notation.

The default transitions are required for the Stateflow diagram to execute. Without the default transition to state `S`, when the Stateflow diagram is awakened, none of the states becomes active. You can detect this situation at run-time by checking for state inconsistencies. See “Animation Controls” on page 12-9 for more information.

Default Transition to a Junction Example

This example shows a default transition to a connective junction.

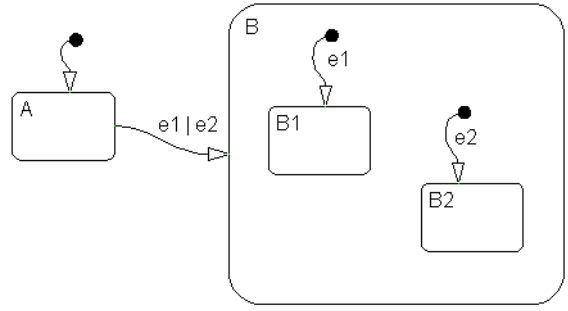


In this example, the default transition to the connective junction defines that upon entering the `Counting` state, the destination is determined by the condition on each transition segment.

See “Default Transition to a Junction Example” on page 4-36 for more information on the semantics of this notation.

Default Transition with a Label Example

The following example shows the labeling of default transitions.



If state A is initially active and either e1 or e2 occurs, the transition from state A to superstate B is valid. The substates B1 and B2 both have default transitions. The default transitions are labeled to specify the event that triggers the transition. If event e1 occurs, the transition A to B1 is valid. If event e2 occurs, the transition A to B2 is valid.

See "Labeled Default Transitions Example" on page 4-39 for more information on the semantics of this notation.

Connective Junctions

A connective junction represents a decision point between alternate transition paths taken for a single transition. See the following topics for more information on connective junctions:

- "What Is a Connective Junction?" on page 3-31 — Defines a connective junction and describes some of the transition constructs it can be used to create.
- "Flow Diagram Notation with Connective Junctions" on page 3-32 — Introduces you to the concept of using connective junctions to create flow diagrams that create the logic of common code structures and provides extensive examples of their use.

What Is a Connective Junction?

The connective junction enables representation of different possible transition paths for a single transition. Connective junctions are used to help represent the following:

- Variations of an `if-then-else` decision construct, by specifying conditions on some or all of the outgoing transitions from the connective junction
- A self-loop transition back to the source state if none of the outgoing transitions is valid
- Variations of a `for` loop construct, by having a self-loop transition from the connective junction back to itself
- Transitions from a common source to multiple destinations
- Transitions from multiple sources to a common destination
- Transitions from a source to a destination based on common events

---

Note An event cannot trigger a transition from a connective junction to a destination state.

See "Connective Junction Examples" on page 4-50 for a summary of the semantics of connective junctions.

Flow Diagram Notation with Connective Junctions

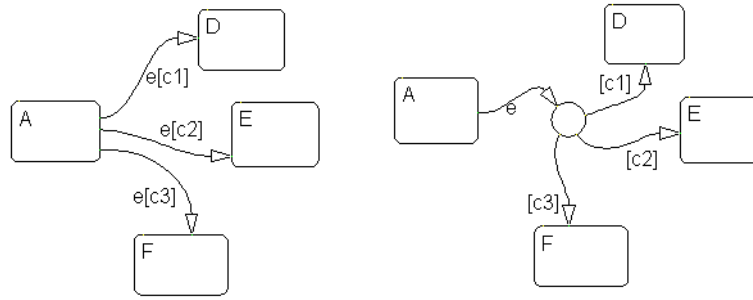
Flow diagram notation uses connective junctions to represent common code structures like `for` loops and `if-then-else` constructs without the use of states. And by reducing the number of states in your Stateflow diagrams, flow diagram notation produces more efficient generated code that helps optimize memory use.

Flow diagram notation employs combinations of the following:

- Transitions to and from connective junctions
- Self-loops to connective junctions
- Inner transitions to connective junctions

Flow diagram notation, states, and state-to-state transitions seamlessly coexist in the same Stateflow diagram. The key to representing flow diagram notation is in the labeling of the transitions (specifically the use of action language) as shown by the following examples.

Connective Junction with All Conditions Specified Example



In the example on the left, if state `A` is active when event `e` occurs, the transition from state `A` to any of states `D`, `E`, or `F` takes place if one of the conditions `[c1]`, `[c2]`, or `[c3]` is met.

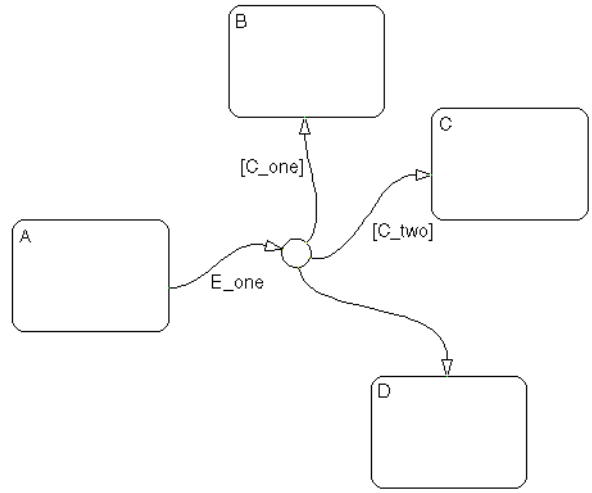
In the equivalent representation on the right, a transition from the source state to a connective junction is labeled by the event. Transitions from the connective junction to the destination states are labeled by the conditions. If state `A` is active when event `e` occurs, the transition from `A` to the connective junction occurs first. The transition from the connective junction to a destination state

follows based on which of the conditions `[c1]`, `[c2]`, or `[c3]` is true. If none of the conditions is true, no transition occurs and state `A` remains active.

See "If-Then-Else Decision Construct Example" on page 4-51 for more information on the semantics of this notation.

Connective Junction with One Unconditional Transition Example

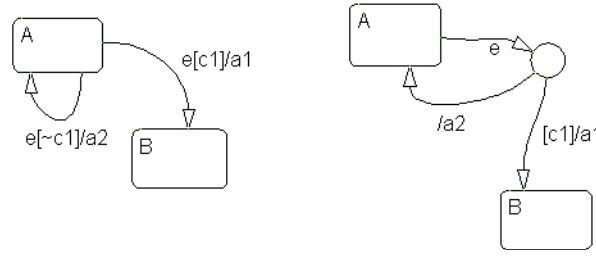
The transition `A` to `B` is valid when `A` is active, event `E_one` occurs, and `[C_one]` is true. The transition `A` to `C` is valid when `A` is active, event `E_one` occurs, and `[C_two]` is true. Otherwise, given `A` is active and event `E_one` occurs, the transition `A` to `D` is valid. If you do not explicitly specify condition `[C_three]`, it is implicit that the transition condition is not `[C_one]` and not `[C_two]`.



See "If-Then-Else Decision Construct Example" on page 4-51 for information on the semantics of this notation.

Connective Junction—Self-Loop Example

In some situations, the transition event occurs but a condition is not met. No transition is taken, but an action is generated. You can represent this situation by using a connective junction or a self-loop transition (transition from state to itself).



In the example on the left, if State `A` is active and event `e` occurs and the condition `[c1]` is met, the transition from `A` to `B` is taken, generating action `a1`. The transition from state `A` to state `A` is valid if event `e` occurs and `[c1]` is not true. In this self-loop transition, the system exits and reenters state `A`, and executes action `a2`.

In the equivalent representation on the right, the use of a connective junction makes it unnecessary to specify the implied condition `[~c1]` explicitly.

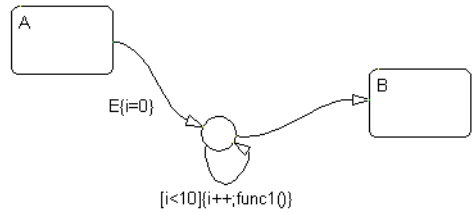
See “Self-Loop Transition Example” on page 4-53 for more information on the semantics of this notation.

Connective Junction and For Loops Example

This example shows a combination of flow diagram notation and state transition notation. Self-loop transitions to connective junctions can be used to represent `for` loop constructs.

In state `A`, event `E` occurs. The transition from state `A` to state `B` is valid if the conditions along the transition path are true. The first segment of the transition does not have a condition, but does have a condition action. The condition action, `{i=0}`, is executed. The condition on the self-loop transition is evaluated as true and the condition actions `{i++;func1()}` execute. The condition actions execute until the condition `[i<10]` is false. The condition actions on both the first segment and the self-loop transition to the connective junction effectively execute a `for` loop (for `i` values 0 to 9 execute `func1()`). The

`for` loop is executed outside the context of a state. The remainder of the path is evaluated. Because there are no conditions, the transition completes at the destination, state `B`.



See "For Loop Construct Example" on page 4-54 for information on the semantics of this notation.

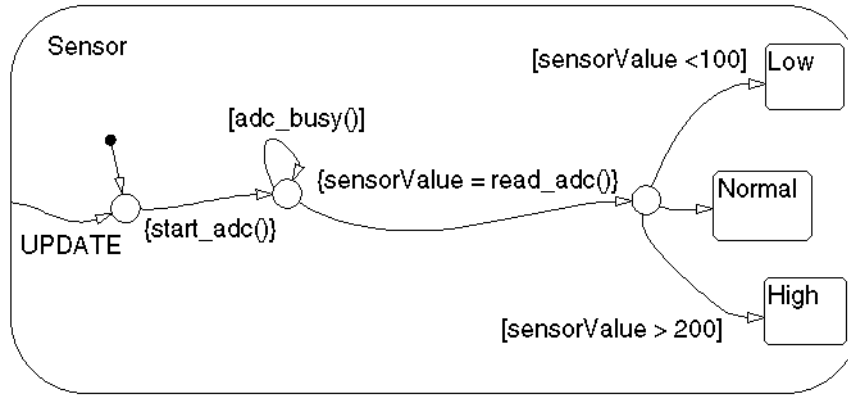
Flow Diagram Notation Example

This example shows a real-world use of flow diagram notation and state transition notation. This Stateflow diagram models an 8-bit analog-to-digital converter (ADC).

Consider the case when state `Sensor.Low` is active and event `UPDATE` occurs. The inner transition from `Sensor` to the connective junction is valid. The next transition segment has a condition action, `{start_adc()}`, which initiates a reading from the ADC. The self-loop on the second connective junction repeatedly tests the condition `[adc_busy()]`. This condition evaluates as true once the reading settles (stabilizes) and the loop completes. This self-loop transition is used to introduce the delay needed for the ADC reading to settle. The delay could have been represented by another state with some sort of counter. Using flow notation in this example avoids an unnecessary use of a state and produces more efficient code.

The next transition segment condition action, `{sensorValue=read_adc()}`, puts the new value read from the ADC in the data object `sensorValue`. The final transition segment is determined by the value of `sensorValue`. If `[sensorValue <100]` is true, the state `Sensor.Low` is the destination. If

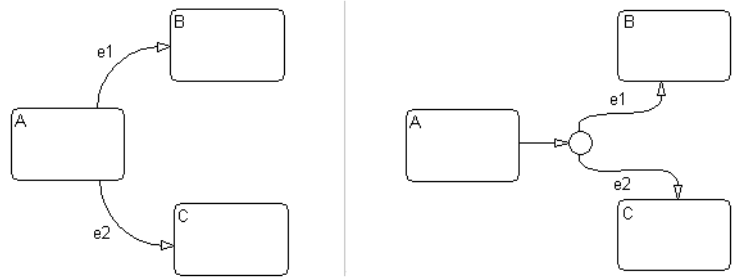
`[sensorValue >200]` is true, the state `Sensor.High` is the destination. Otherwise, state `Sensor.Normal` is the destination state.



See "Flow Diagram Notation Example" on page 4-55 for information on the semantics of this notation.

Connective Junction from a Common Source to Multiple Destinations Example

Transitions `A` to `B` and `A` to `C` share a common source state `A`. An alternative representation uses one arrow from `A` to a connective junction, and multiple arrows labeled by events from the junction to the destination states `B` and `C`.



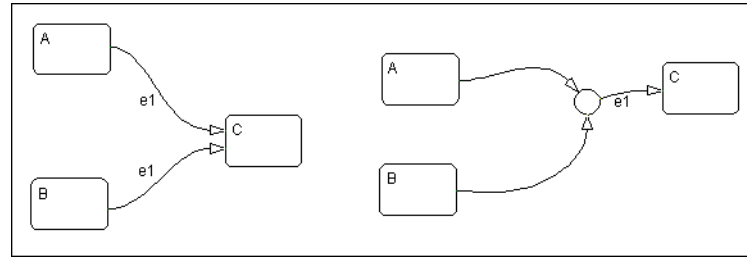
See "Transitions from a Common Source to Multiple Destinations Example" on page 4-57 for information on the semantics of this notation.

Connective Junction Common Events Example

Suppose, for example, that when event `e1` occurs, the system, whether it is in state `A` or `B`, transfers to state `C`. Suppose that transitions `A` to `C` and `B` to `C` are triggered by the same event `e1`, so that both destination state and trigger event are common to the transitions. There are three ways to represent this:

- By drawing transitions from `A` and `B` to `C`, each labeled with `e1`
- By placing `A` and `B` in one superstate `S`, and drawing one transition from `S` to `C`, labeled with `e1`
- By drawing transitions from `A` and `B` to a connective junction, then drawing one transition from the junction to `C`, labeled with `e1`

This Stateflow diagram shows the simplification using a connective junction.



See “Transitions from a Source to a Destination Based on a Common Event Example” on page 4-60 for information on the semantics of this notation.

History Junctions

History junctions record the previously active state of the state in which they are resident. See the following sections for information on history junctions:

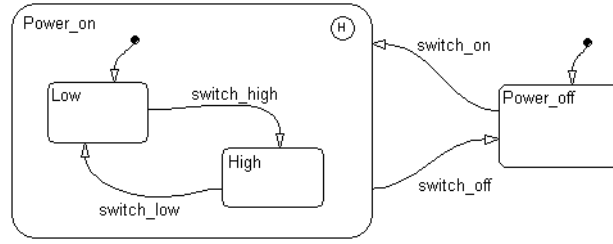
- "What Is a History Junction?" on page 3-38 — Presents a defining description and example of a history junction.
- "History Junctions and Inner Transitions" on page 3-39 — Shows how an inner transition to a history junction can immediately exit and reenter a state as a special behavior.

What Is a History Junction?

A history junction is used to represent historical decision points in the Stateflow diagram. The decision points are based on historical data relative to state activity. Placing a history junction in a superstate indicates that historical state activity information is used to determine the next state to become active. The history junction applies only to the level of the hierarchy in which it appears.

Use of History Junctions Example

The following example uses a history junction:



Superstate `Power_on` has a history junction and contains two substates. If state `Power_off` is active and event `switch_on` occurs, the system could enter either `Power_on.Low` or `Power_on.High`. The first time superstate `Power_on` is entered, substate `Power_on.Low` is entered because it has a default transition. At some point afterward, if state `Power_on.High` is active and event `switch_off` occurs, superstate `Power_on` is exited and state `Power_off` becomes active. Then event `switch_on` occurs. Since `Power_on.High` was the last active state, it

becomes active again. After the first time `Power_on` becomes active, the choice between entering `Power_on.Low` or `Power_on.High` is determined by the history junction.

See "Default Transition and a History Junction Example" on page 4-37 for more information on the semantics of this notation.

History Junctions and Inner Transitions

By specifying an inner transition to a history junction, you can specify that, based on a specified event and/or condition, the active state is to be exited and then immediately reentered.

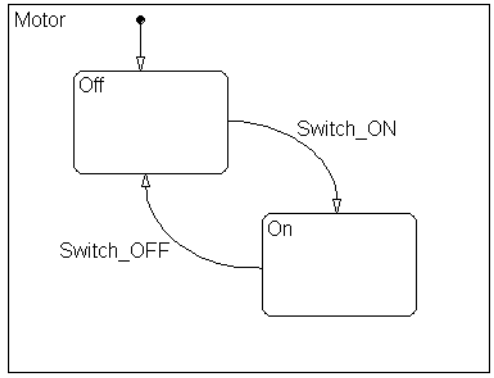
See "Using an Inner Transition to a History Junction" on page 3-25 for an example of this notation.

See "Inner Transition to a History Junction Example" on page 4-48 for more information on the semantics of this notation.

Boxes

You use boxes to graphically organize your diagram. Beyond this organizational use, boxes contribute little to how Stateflow diagrams execute.

The following is an example of a Stateflow Box object:

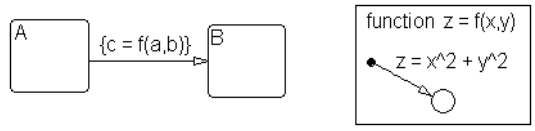


In this example, a box labeled `Motor` groups all the objects needed to control a simple motor. There can be many more objects displayed on the Stateflow chart along with this box, but now everything needed to control the motor is kept separate.

Graphical Functions

A *graphical function* is a function defined graphically by a flow graph that provides convenience and power to Stateflow action language.

The following example shows a graphical function side by side in a Stateflow diagram with the transition that calls it:



In this example the function `z = f(x,y)` is called in the condition action of the transition from state A to state B. The function is defined using symbols that are valid only within the function itself. The function is called using data objects available to states A and B and their parent states (if any).

Graphical functions are similar to textual functions such as MATLAB and C functions. Like C and MATLAB functions, graphical functions can accept arguments and return results. Like C and MATLAB functions, you invoke graphical functions in transition and state actions. Unlike C and MATLAB functions, however, graphical functions are full-fledged Stateflow graphical objects.

You use the Stateflow editor to create them and they reside in your Stateflow model along with the diagrams that invoke them. This makes graphical functions easier to create, access, and manage than textual functions, whose creation requires external tools and whose definitions reside separately from the model.

Action Language

This section introduces you to language included in state and transition labels that executes with the execution of the state or transition. See the following topics:

- "What Is Action Language?" on page 3-42 — Describes action language and provides a general example of its use.
- "Action Language Categories Example" on page 3-43 — Gives you an overview of the different categories of action language through an example Stateflow diagram.
- "Action Language Notation" on page 3-43 — Describes action language by referring to topics that define the labels of states and transitions in Stateflow.
- "Action Language Semantics" on page 3-44 — Describes the interaction of different categories of action language on a state or transition.

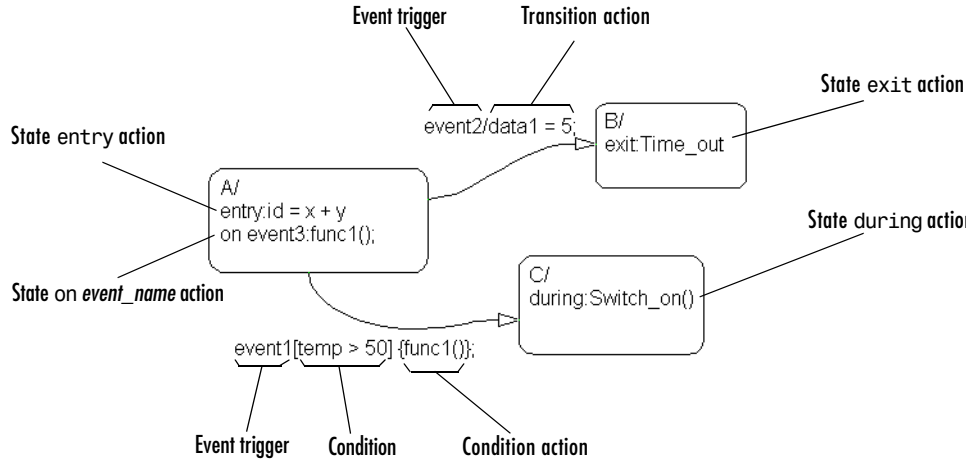
What Is Action Language?

Action language defines the categories of actions you can specify using the notation of state and transition labels described in the following topics:

Stateflow attaches actions to a state or transition through its label. The actions for states and transitions are divided into different types of action using label notation. Transitions have event triggers, conditions, condition actions, and transition actions. States can have `entry`, `during`, `exit`, and, `on` *`event_name`* actions.

Action Language Categories Example

The following diagram shows examples of the different categories of actions for states and transitions:



The actions in action language can be broadcast events, condition statements, function calls, variable assignments and operations, etc. Many are very similar to statements in C or MATLAB. See Chapter 7, "Action Language," for a description of allowable actions in the action language of Stateflow.

Action Language Notation

Action language for states and transitions is included in the label for state or transition to which the action language applies. See the following topics for detailed information on action language notation:

- "State Label Notation" on page 3-10 — Describes action language for states, which is included in the label for a state
- "Transition Label Notation" on page 3-15 — Describes action language for transitions which is included in the label of a transition.

Action Language Semantics

The rules for interaction between the different Stateflow notations are referred to as Stateflow semantics. See Chapter 4, "Stateflow Semantics," for a more detailed understanding of Stateflow semantics (behavior).

For the semantics of state action language, see the following topics:

- "Executing a State" on page 4-13 — Describes the execution of states that includes state action language.
- "State Execution Example" on page 4-16 — Gives an example of state execution that includes state action language in generalized format.

For the semantics of transition action language, see the following topics:

- "Executing a Transition" on page 4-6 — Describes the execution of transitions that include transition action language.
- "Label Format for a State-to-State Transition Example" on page 4-23 — Gives an example of transition execution that includes transition action language in generalized format.
- "Label Format for Transition Segments Example" on page 4-50 — Gives an example of transition execution through transition segments to junctions that includes transition action language in generalized format.

4

Stateflow Semantics

Stateflow semantics describe how the notation in Stateflow charts is interpreted and implemented into a behavior. Knowledge of Stateflow semantics is important to make sound Stateflow diagram design decisions for code generation. Different notations result in different behavior during simulation and generated code execution. This chapter contains the following sections:

Executing an Event (p. 4-3)	Describes the behavior of events that drive Stateflow chart execution.
Executing a Chart (p. 4-5)	Describes how charts become active, execute, and become inactive.
Executing a Transition (p. 4-6)	Describes the processes for grouping, ordering, and executing a transition.
Executing a State (p. 4-13)	Describes how states become active, execute, and become inactive.
Early Return Logic for Event Broadcasts (p. 4-18)	Describes the logic employed when events interrupt the normal execution behavior of Stateflow charts.
Semantic Examples (p. 4-21)	A list of the semantic (behavioral) examples provided in the remainder of this chapter.
Transitions to and from Exclusive (OR) States Examples (p. 4-23)	Examples that describe the behavior of transitions that exit and enter exclusive (OR) states.
Condition Action Examples (p. 4-29)	Examples that describe the behavior of Stateflow diagrams using condition actions.
Default Transition Examples (p. 4-35)	Examples that describe the behavior of Stateflow diagrams using default transitions.
Inner Transition Examples (p. 4-41)	Examples that describe the behavior of Stateflow diagrams using inner transitions.
Connective Junction Examples (p. 4-50)	Example that describe the behavior of Stateflow diagrams using connective junctions.

Event Actions in a Superstate Example (p. 4-63)	Example that describe the behavior of Stateflow diagrams using event actions.
Parallel (AND) State Examples (p. 4-65)	Example Stateflow diagrams demonstrating the behavior of parallel (AND) states.
Directed Event Broadcasting Examples (p. 4-77)	Example Stateflow diagrams demonstrating how to use directed event broadcasting.

Executing an Event

A Stateflow chart executes only in response to an event. This occurs on two levels. First, Simulink updates the chart, which awakens it for execution. Second, once the chart is awakened, it continues to respond to events until there are no more events. The chart then goes to sleep. When another event occurs, the chart is awakened (from sleep) to respond to the event.

Because Stateflow runs on a single thread, actions that take place based on an event are atomic to that event. This means that all activity caused by the event in the chart is completed before returning to whatever activity was taking place prior to reception of the event. Once action is initiated by an event, it is completed unless interrupted by an early return.

See the following topics to continue with the behavior of events:

- "Sources for Stateflow Events" on page 4-3 — Describes the sources for events that drive Stateflow diagrams.
- "Processing Events" on page 4-4 — Describes how Stateflow diagrams handle events.

Sources for Stateflow Events

Stateflow charts are awakened by Simulink events. From the Stateflow perspective, this is an event like any other event, with the exception that it comes from Simulink. After the chart is made active, it can respond to the occurrence of this event in its action language.

You can also use events to control the processing of your Stateflow diagrams by broadcasting events in the action language associated with states and transitions in the chart itself. For the mechanics of broadcasting events in action language, see "Event Broadcasting" on page 7-62. For examples using event broadcasting and directed event broadcasting, see the following:

- "Condition Actions to Broadcast Events to Parallel (AND) States Example" on page 4-32
- "Cyclic Behavior to Avoid with Condition Actions Example" on page 4-33
- "Event Broadcast State Action Example" on page 4-65
- "Event Broadcast Transition Action with a Nested Event Broadcast Example" on page 4-69

- "Event Broadcast Condition Action Example" on page 4-72
- "Directed Event Broadcasting" on page 7-64

Before you can broadcast events in the action language of a Stateflow chart, you must first add them. You can do this in the **Add** menu of the Stateflow diagram editor or through the Explorer. Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event's parent that determines who can trigger on the event (has receive rights). See the **Name** and **Parent** fields for an event in "Setting Event Properties" on page 6-4 for more information.

Processing Events

When an event occurs, it is processed from the top or root of the Stateflow diagram down through the hierarchy of the diagram. At each level in the hierarchy, any `during` and `on` *`event_name`* actions for the active state are executed and completed and then a check for the existence of a valid explicit or implicit transition among the children of the state is conducted. The examples in this chapter demonstrate the top-down processing of events.

All events, with the exception of the output edge trigger to Simulink (see the following note), have the following execution in a Stateflow diagram:

1. If the *receiver* of the event is active, then it is executed (see "Executing an Active Chart" on page 4-5 and "Executing an Active State" on page 4-15). (The event *receiver* is the parent of the event unless the event was explicitly directed to a *receiver* using the `send()` function.)

2. If the receiver of the event is not active, nothing happens.

3. After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

   For an understanding of early return logic, see "Early Return Logic for Event Broadcasts" on page 4-18.

Note Output edge trigger event execution in Simulink is equivalent to toggling the value of an output data value between 1 and 0. It is not treated as a Stateflow event. See "Defining Edge-Triggered Output Events" on page 8-19.

Executing a Chart

A Stateflow chart executes when it is triggered by an event from Simulink. Like all events, this event is processed top down in the updated chart. See “Executing an Event” on page 4-3.

A chart is inactive when it is first triggered by an event from the Simulink model and has no active states within it. After the chart executes and completely processes its initial trigger event from the Simulink model, it exits to the model and goes to sleep, but still remains active. A sleeping chart has active states within it, but no events to process. When Simulink triggers the chart the next time, it is an active but sleeping chart.

Executing an Inactive Chart

When a chart is inactive and first triggered by an event from Simulink, it first executes its set of default flow graphs (see “Executing a Set of Flow Graphs” on page 4-7). If this does not cause an entry into a state and the chart has parallel decomposition, then each parallel state is entered (see “Entering a State” on page 4-13).

If executing the default flow paths does not cause state entry, a state inconsistency error occurs.

Executing an Active Chart

After a chart has been triggered the first time by the Simulink model, it is an active chart. When it receives another event from Simulink, it executes again as an active chart. If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active children are executed. Parallel states are executed in the same order that they are entered.

Executing a Transition

Transitions play a large role in defining the animation or execution of a system. If your chart has exclusive (OR) states, its execution begins with the default transitions that point to the first active states in your chart.

Transitions have sources and destinations; thus, any actions associated with the sources or destinations are related to the transition that joins them. The type of the source and destination is equally important to define the semantics.

See the following topics:

- "Transition Flow Graph Types" on page 4-6 — Defines the groups used for ordering transitions in order to select a valid transition for execution.

  Outgoing transitions from an active state are first grouped according to their flow graph type.
- "Executing a Set of Flow Graphs" on page 4-7 — Describes the process of selecting a valid transition from a group.

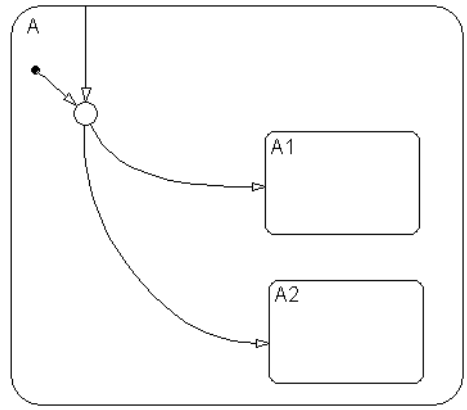
  The transitions in each flow graph group are checked for a valid, executable transition, starting with the highest priority group, and ending with the lowest priority group, until a valid transition is found.
- "Ordering Single Source Transitions" on page 4-8 — Describes the process through which transitions with the same source are ordered before selecting a valid transition for execution.

Transition Flow Graph Types

Before transitions are executed for an active state or for a chart, they are grouped by the following types:

- Default flow graphs are all default transition segments that start with the same parent.
- Inner flow graphs are all transition segments that originate on a state and reside entirely within that state.
- Outer flow graphs are all transition segments that originate on the respective state but reside at least partially outside that state.

Each set of flow graphs includes other transition segments connected to a qualifying transition segment through junctions and transitions. Consider the following example:



In this example, state `A` has both an inner and a default transition that connect to a junction with outgoing transitions to states `A.A1` and `A.A2`. If state A is active, its set of inner flow graphs includes the inner transition and the outgoing transitions from the junction to state `A.A1` and `A.A2`. In addition, state `A`'s set of default flow graphs includes the default transition to the junction along with the two outgoing transitions from the junction to state `A.A1` and `A.A2`. In this case, the two outgoing transition segments from the junction become members of more than one flow graph type.

Executing a Set of Flow Graphs

Each flow graph group is executed in the order of group priority until a valid transition is found. The default transitions group is executed first, followed by the inner transitions group and then the outer transitions group. Each flow graph group is executed with the following procedure.

1 Order the group's transition segments for the active state.

An active state can have several possible outgoing transitions. These are ordered before checking them for a valid transition. See "Ordering Single Source Transitions" on page 4-8.

2 Select the next transition segment in the set of ordered transitions.

3 Test the transition segment for validity.

4 If the segment is invalid, go to step 2.

5 If the destination of the transition segment is a state, do the following:

   a No more transition segments are tested and a transition path is formed by including the transition segment from each preceding junction back to the starting transition.

   b The states that are the immediate children of the parent of the transition path are exited (see "Executing an Active State" on page 4-15).

   c The transition action for the final transition segment of the full transition path is executed.

   d The destination state is entered (see "Entering a State" on page 4-13).

6 If the destination is a junction with no outgoing transition segments, do the following:

   a Testing stops without any states being exited or entered.

7 If the destination is a junction with outgoing transition segments, repeat step 1 for the set of outgoing segments from the junction.

8 After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the backup segment. The set of flow graphs is done executing when all starting transitions have been tested.

Ordering Single Source Transitions

Transitions from a single source are ordered for testing according to the following three sorting guidelines, which appear in order of their precedence (first step is highest priority):

1 Endpoint Hierarchy — Transitions whose end points are attached to higher hierarchical levels are placed first in testing order. See the topic "Ordering by Hierarchy" on page 4-9.

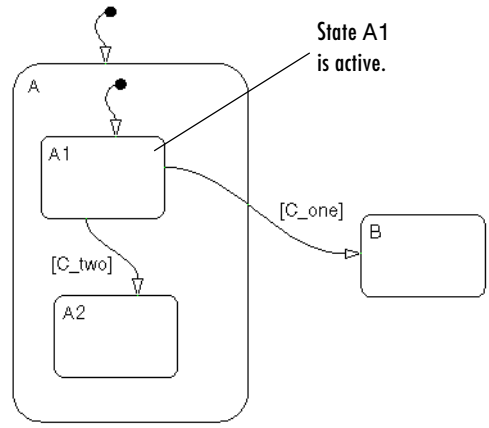
2 Label — Transitions are ordered for testing according to the types of action language present in their labels. See "Ordering by Label" on page 4-9.

3 Angular Surface Position of Transition Source — Transitions are ordered for testing based on the angular position of the transition source on the surface of the originating object. See "Ordering by Angular Surface Position of Source" on page 4-10.

Note Do not design your Stateflow diagram based on the expected execution order of transitions.

Ordering by Hierarchy

Transitions are evaluated in a top-down manner based on hierarchy. In the following example, an event occurs while state A1 is active.



Because state B is a sibling of state A and at a higher hierarchical level than state A2, a sibling of A1, the transition from state A1 to state B takes precedence over the transition from state A1 to state A2.

Ordering by Label

Transitions of equal endpoint hierarchical level are evaluated based on their labels, in the following order of precedence:

1 Labels with events and conditions

2 Labels with events

3 Labels with conditions

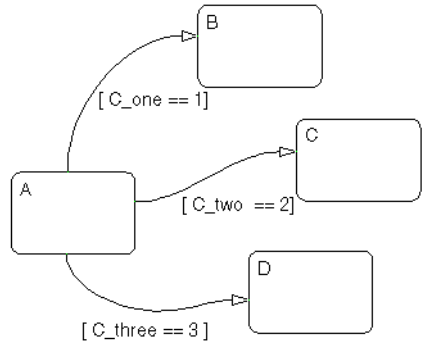
4 No label

The following example demonstrates ordering of single source transitions by the angular surface position of the source.

Ordering by Angular Surface Position of Source

Equivalent transitions (based on their labels and the hierarchy of their source and endpoints) are ordered based on the angular position on the surface of the source object for the outgoing transitions.

Multiple outgoing transitions from states that are of equivalent label and source and end point hierarchy priority are evaluated in a clockwise progression starting at the upper left corner of the source state.



In this example, the transitions are of equivalent label priority. The conditions [`C_two` == 2] and [`C_three` == 3] are both true and [`C_one` = = 1] is false. Also, the hierarchical level of the endpoint of each transition is the same because all the states in the example are siblings.

The outgoing transitions from state A in the preceding diagram are evaluated in the following order:

1 The angular position of the source point for the transition from state `A` to state `B` is 12 o'clock.

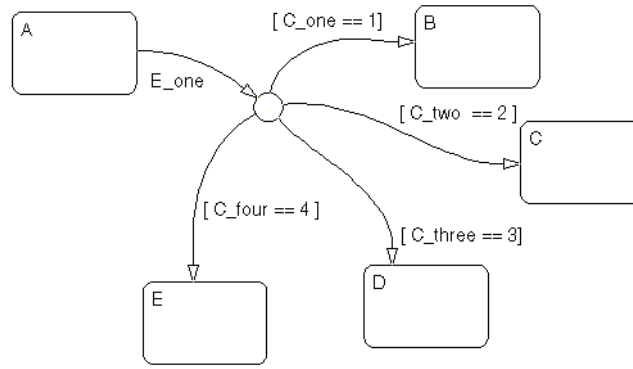
   Since the condition [`C_one == 1`] is false, this transition is not valid.

2 The angular position of the source point for the transition from state `A` to state `C` is 2 o'clock.

   Since the condition [C_two == 2] is true, this transition is valid and is taken.

The angular position of the source point for the transition from state `A` to state `D` is 6 o'clock. This transition, even though it is valid, is not evaluated since the previous transition was taken.

Multiple outgoing transitions from junctions that are of equivalent label priority are evaluated in a clockwise progression starting from a twelve o'clock position on the junction.



In this example, the transitions are of equivalent label priority. Also, the conditions [`C_three == 3`]and [`C_four == 4`] are both true. Given that, the outgoing transitions from the junction are evaluated in this order:

1 The transition to state B is evaluated. Since the condition [`C_one == 1`] is false, this transition is not valid.

2 The transition to state C is evaluated. Since the condition [`C_two == 2`] is false, this transition is not valid.

3 The transition to state D is evaluated. Since the condition [`C_three` == 3] is true, this transition is valid and is taken.

Since the transition to D is taken, the transition to state E is not evaluated.

Executing a State

States are either active or inactive. The following sections describe the stages of state execution that take place between becoming active and becoming inactive:

- "Entering a State" on page 4-13 — Describes the execution involved when a transition causes a state to be entered.
- "Executing an Active State" on page 4-15 — Describes the execution of a state when it receives an event that does not result in an outgoing transition from that state to another.
- "Exiting an Active State" on page 4-15 — Describes the execution involved when a transition causes a state to be exited.
- "State Execution Example" on page 4-16 — Gives a detailed description of an example of state execution activities during the execution of a Stateflow diagram.

Entering a State

A state is entered (becomes active) in one of the following ways:

- Its boundaries are crossed by an incoming executed transition.
- Its boundary terminates the arrow end of an incoming transition.
- It is the parallel state child of an activated state.

A state performs its `entry` action (if specified) when it becomes active. The state is marked active before its `entry` action is executed and completed.

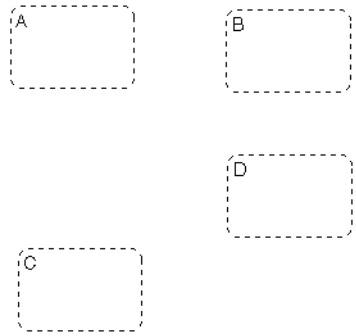
The execution steps for entering a state are as follows:

1. If the parent of the state is not active, perform steps 1 through 4 for the parent first.

2. If this is a parallel state, check to make sure that all sibling parallel states with a higher execution order are active. If not, perform all entry steps for these states first in the appropriate order of entry.

   Parallel (AND) states are ordered for entry based on their vertical top-to-bottom position in the diagram editor. Parallel states that occupy the same vertical level are ordered for entry from left to right. In the following

example, parallel states A and B are aligned at the same vertical level while states A and C and states B and D are aligned at the same horizontal position. Based on their top-down positions in the diagram editor, the order of execution for these states is A or B, then D and C. Because A is left of B, A is evaluated first and the order of entry is A, B, D, C.



3. Mark the state active.
4. Perform any entry actions.
5. Enter children, if needed:
   a. Execute the default flow paths for the state unless it contains a history junction.
   b. If the state contains a history junction and there is an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child.
   c. If this state has children that are parallel states (parallel decomposition), perform entry steps 1 to 5 for each state according to its entry order.
6. If this is a parallel state, perform all entry actions for the sibling state next in entry order if one exists.
7. If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.
8. The chart goes to sleep.

Executing an Active State

Active states that receive an event that does not result in an exit from that state execute a `during` action to completion if a `during` action is specified for that state. An `on` *`event_name`* action executes to completion when the event specified, *`event_name`*, occurs and that state is active. An active state executes its `during` and `on` *`event_name`* actions before processing any of its children's valid transitions. `During` and `on` *`event_name`* actions are processed based on their order of appearance in the state label.

The execution steps for executing a state that receives an event while it is active are as follows:

1. The set of outer flow graphs is executed (see "Executing a Set of Flow Graphs" on page 4-7).

   If this causes a state transition, execution of the state stops.

Note This step is never required for parallel states.

---

2. `During` actions and valid `on` *`event name`* actions are preformed.

3. The set of inner flow graphs is executed. If this does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Exiting an Active State

A state is exited (becomes inactive) in one of the following ways:

- Its boundary is the origin of an outgoing executed transition.
- Its boundary is crossed by an outgoing executed transition.
- It is a parallel state child of an activated state.

A state performs its `exit` action (if specified) before it becomes inactive. The state is marked inactive after the `exit` action has executed and completed.

The execution steps for exiting a state are as follows:

1 If this is a parallel state, and one of its sibling states was entered before this state, exit the siblings starting with the last-entered and progressing in reverse order to the first-entered. See step 2 of "Entering a State" on page 4-13.

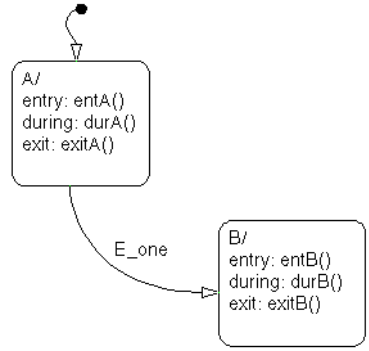
2 If there are any active children, perform the exit steps on these states in the reverse order they were entered.

3 Perform any exit actions.

4 Mark the state as inactive.

State Execution Example

The following example demonstrates the execution semantics (behavior) of event reactive behavior by active and inactive states.



Inactive Diagram Event Reaction

Initially the Stateflow diagram and its states are inactive. This is the semantic sequence that follows an event:

1 An event occurs and the Stateflow diagram is awakened.

2 The Stateflow diagram checks to see if there is a valid transition as a result of the event.

A valid default transition to state `A` is detected.

3 State A is marked active.

4 State A `entry` actions (`entA()`) execute and complete.

5 The Stateflow diagram goes back to sleep.

Sleeping Diagram Event Reaction

The Stateflow diagram is now asleep and waiting to be awakened by another event.

1 Event `E_one` occurs and the Stateflow diagram is awakened.

   State A is active from the preceding steps 1 to 5.

2 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition is detected from state A to state B.

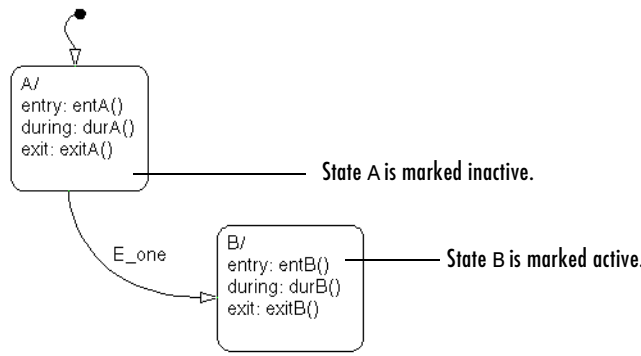
3 State A `exit` actions (`exitA()`) execute and complete.

4 State A is marked inactive.

5 State B is marked active.

6 State B `entry` actions (`entB()`) execute and complete.

7 The Stateflow diagram goes back to sleep, to be awakened by the next event.



Early Return Logic for Event Broadcasts

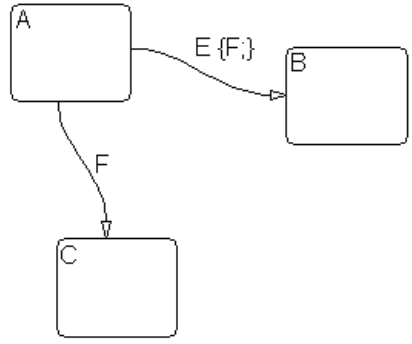
Stateflow employs early return logic in order to satisfy conflicts with proper diagram behavior that result from event broadcasts in state or transition actions.

The following statements are primary axioms of proper Stateflow behavior:

1. Whenever a state is active, its parent should also be active.
2. A state (or chart) with exclusive (OR) decomposition must never have more than one active child.
3. If a parallel state is active, siblings with higher priority (higher graphical position in the Stateflow diagram) must also be active.

Because Stateflow runs on a single thread, when it receives an event it must interrupt its current activity to process all activity resulting from the broadcast event before returning to its original activity. However, activity resulting from an event broadcast can conflict with the current activity, giving rise to the event broadcast. This conflict is resolved through early return logic.

The need for early return logic is best illustrated with an example like the following:



In this example, assume that state `A` is initially active. An event, `E`, occurs and the following behavior is expected:

1. The Stateflow diagram root checks to see if there is a valid transition out of the active state `A` as a result of event `E`.
2. A valid transition terminating in state `B` is found.
3. The condition action of the valid transition executes and broadcasts event F.

   Stateflow must now interrupt the anticipated transition from `A` to `B` and take care of any behavior resulting from the broadcast of the event `F` before continuing with the transition from `A` to `B`.
4. The Stateflow diagram root checks to see if there is a valid transition out of the active state `A` as a result of event `F`.
5. A valid transition terminating in state `C` is found.
6. State `A` executes its `exit` action.
7. State `A` is marked inactive.
8. State `C` is marked active.
9. State `C` executes and completes its `entry` action.

State `C` is now the only active child of its chart. Stateflow cannot return to the transition from state `A` to state `B` and continue after the condition action that broadcast event `F` (step 3). First, its source, state `A`, is no longer active. Second, if Stateflow were to allow the transition, state `B` would become the second active child of the chart. This violates the second Stateflow axiom that a state (or chart) with exclusive (OR) decomposition can never have more than one active child. Consequently, early return logic is employed, and the transition from state `A` to state `B` is halted.

In order to maintain primary axiomatic behavior in Stateflow diagrams, Stateflow employs early return logic for event broadcasts in each of its action types as follows:

Action Type	Early Return Logic
Entry	If the state is no longer active at the end of the event broadcast, any remaining steps for entering a state are not performed.
Exit	If the state is no longer active at the end of the event broadcast, any remaining exit actions or steps in transitioning from state to state are not performed.
During	If the state is no longer active at the end of the event broadcast, any remaining steps in the execution of active state are not performed.
Condition	If the origin state of the inner or outer flow graph or parent state of the default flow graph is no longer active at the end of the event broadcast, the remaining steps in the execution of the set of flow graphs are not performed.
Transition	If the parent of the transition path is not active or if that parent has an active child, the remaining transition actions and state entry are not performed.

Semantic Examples

The following is a list of the examples provided to demonstrate the semantics (behavior) of Stateflow charts.

"Transitions to and from Exclusive (OR) States Examples" on page 4-23

- "Transitioning from State to State with Events Example" on page 4-24
- "Transitioning from a Substate to a Substate with Events Example" on page 4-27

"Condition Action Examples" on page 4-29

- "Condition Action Example" on page 4-29
- "Condition and Transition Actions Example" on page 4-31
- "Condition Actions in For Loop Construct Example" on page 4-32
- "Condition Actions to Broadcast Events to Parallel (AND) States Example" on page 4-32
- "Cyclic Behavior to Avoid with Condition Actions Example" on page 4-33

"Default Transition Examples" on page 4-35

- "Default Transition in Exclusive (OR) Decomposition Example" on page 4-35
- "Default Transition to a Junction Example" on page 4-36
- "Default Transition and a History Junction Example" on page 4-37
- "Labeled Default Transitions Example" on page 4-39

"Inner Transition Examples" on page 4-41

- "Processing One Event in an Exclusive (OR) State" on page 4-41
- "Processing a Second Event in an Exclusive (OR) State" on page 4-42
- "Processing a Third Event in an Exclusive (OR) State" on page 4-43
- "Processing the First Event with an Inner Transition to a Connective Junction" on page 4-45
- "Processing a Second Event with an Inner Transition to a Connective Junction" on page 4-46
- "Inner Transition to a History Junction Example" on page 4-48

"Connective Junction Examples" on page 4-50

- "If-Then-Else Decision Construct Example" on page 4-51
- "Self-Loop Transition Example" on page 4-53
- "For Loop Construct Example" on page 4-54
- "Flow Diagram Notation Example" on page 4-55
- "Transitions from a Common Source to Multiple Destinations Example" on page 4-57
- "Transitions from Multiple Sources to a Common Destination Example" on page 4-59
- "Transitions from a Source to a Destination Based on a Common Event Example" on page 4-60

"Event Actions in a Superstate Example" on page 4-63

- "Event Actions in a Superstate Example" on page 4-63

"Parallel (AND) State Examples" on page 4-65

- "Event Broadcast State Action Example" on page 4-65
- "Event Broadcast Transition Action with a Nested Event Broadcast Example" on page 4-69
- "Event Broadcast Condition Action Example" on page 4-72

Directed Event Broadcasting

- "Directed Event Broadcast Using send Example" on page 4-77
- "Directed Event Broadcasting Using Qualified Event Names Example" on page 4-79

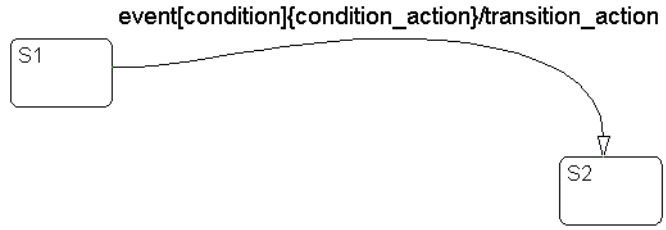
Transitions to and from Exclusive (OR) States Examples

The following examples demonstrate the use of state-to-state transitions to and from exclusive (OR) states in Stateflow:

- "Label Format for a State-to-State Transition Example" on page 4-23 — Shows the semantics for a transition with a general label format.
- "Transitioning from State to State with Events Example" on page 4-24 — Shows the behavior of a simple transition focusing on the implications of whether states are active or inactive for handling events.
- "Transitioning from a Substate to a Substate with Events Example" on page 4-27 — Shows the behavior of a transition from an OR substate to an OR substate.

Label Format for a State-to-State Transition Example

The following example shows the general label format for a transition entering a state:



Execution of the above transition occurs as follows:

1. When an event occurs, state S1 checks for an outgoing transition with a matching event specified.
2. If a transition with a matching event is found, the condition for that transition (`[condition]`) is evaluated.

3 If the condition `condition` evaluates to true, the condition action `condition_action` (`{condition_action}`) is executed.

4 If the destination state is determined to be a valid destination, the transition is taken.

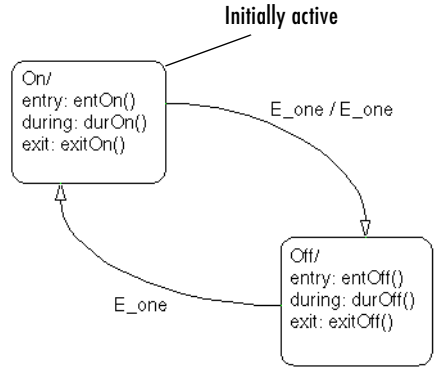
5 State S1 is exited.

6 The transition action `transition_action` is executed when the transition is taken.

7 State S2 is entered.

Transitioning from State to State with Events Example

The following example shows the behavior of a simple transition focusing on the implications of whether states are active or inactive.



Processing of a First Event

Initially the Stateflow diagram is asleep. State `On` and state `Off` are OR states. State `On` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

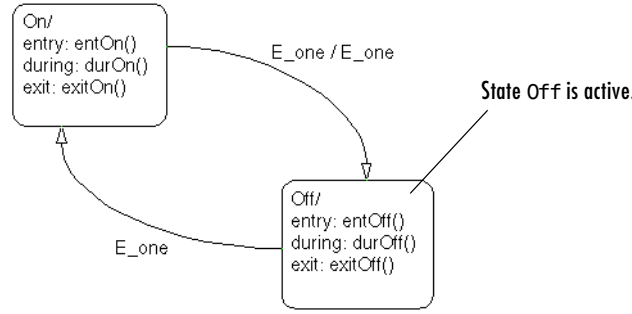
1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `On` to state `Off` is detected.
2. State `On` `exit` actions (`ExitOn()`) execute and complete.
3. State `On` is marked inactive.
4. The event `E_one` is broadcast as the transition action.

   This second event `E_one` is processed, but because neither state is active, it has no effect. Had a valid transition been possible as a result of the broadcast of `E_one`, the processing of the first broadcast of `E_one` would be preempted by the second broadcast of `E_one`. See "Early Return Logic for Event Broadcasts" on page 4-18.
5. State `Off` is marked active.
6. State `Off` `entry` actions (`entOff()`) execute and complete.
7. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event `E_one` when state `On` is initially active.

Processing of a Second Event

Using the same example, what happens when the next event, `E_one`, occurs while state `Off` is active?



Again, initially the Stateflow diagram is asleep. State `Off` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram with the following steps:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`.

   A valid transition from state `Off` to state `On` is detected.

2. State `Off` `exit` actions (`exitOff()`) execute and complete.

3. State `Off` is marked inactive.

4. State `On` is marked active.

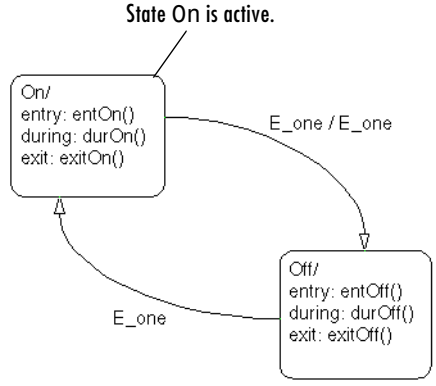
5. State `On` `entry` actions (`entOn()`) execute and complete.

6. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with the second event `E_one` when state `Off` is initially active.

Processing of a Third Event

Using the same example, what happens when a third event, `E_two`, occurs?



Notice that the event E_two is not used explicitly in this example. However, its occurrence (or the occurrence of any event) does result in behavior. Initially, the Stateflow diagram is asleep and state `On` is active.

1. Event `E_two` occurs and awakens the Stateflow diagram.

   Event `E_two` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

2. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is none.

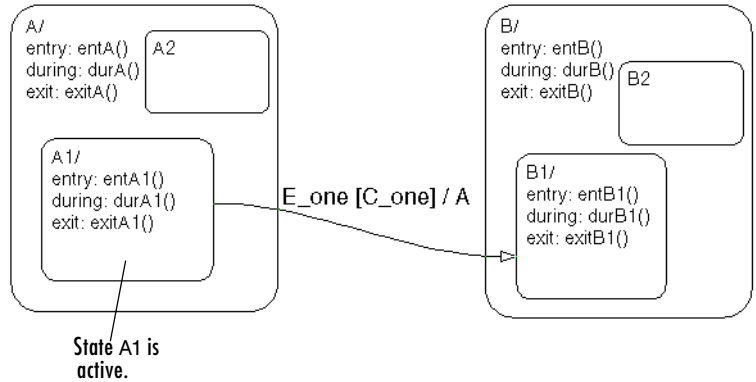
3. State `On` `during` actions (`durOn()`) execute and complete.

4. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event `E_two` when state `On` is initially active.

Transitioning from a Substate to a Substate with Events Example

This example shows the behavior of a transition from an OR substate to an OR substate.



Initially the Stateflow diagram is asleep. State A.A1 is active. Event E_one occurs and awakens the Stateflow diagram. Condition C_one is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A.A1 to state B.B1. (Condition C_one is true.)
2. State A during actions (durA()) execute and complete.
3. State A.A1 exit actions (exitA1()) execute and complete.
4. State A.A1 is marked inactive.
5. State A exit actions (exitA()) execute and complete.
6. State A is marked inactive.
7. The transition action, A, is executed and completed.
8. State B is marked active.
9. State B entry actions (entB()) execute and complete.
10. State B.B1 is marked active.
11. State B.B1 entry actions (entB1()) execute and complete.
12. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

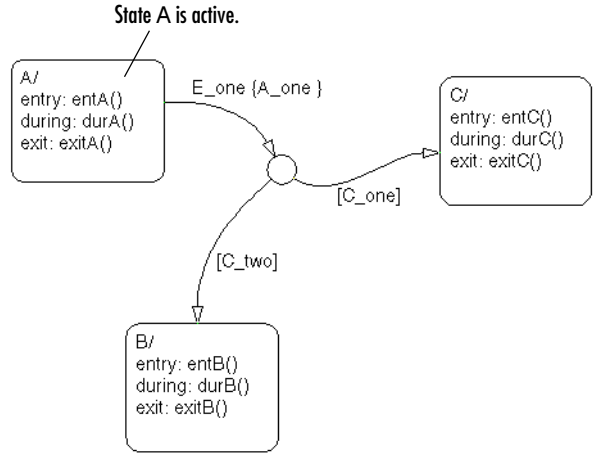
Condition Action Examples

The following examples demonstrate the use of condition actions in Stateflow:

- "Condition Action Example" on page 4-29 — Shows the behavior of a simple condition action in a multiple segment transition.
- "Condition and Transition Actions Example" on page 4-31 — Shows the behavior of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.
- "Condition Actions in For Loop Construct Example" on page 4-32 — Shows the use of a condition action and connective junction to create a for loop construct.
- "Condition Actions to Broadcast Events to Parallel (AND) States Example" on page 4-32 — Shows the use of condition actions used to broadcast events immediately to parallel (AND) states.
- "Cyclic Behavior to Avoid with Condition Actions Example" on page 4-33 — Shows a notation to avoid when using event broadcasts as condition actions because the semantics result in cyclic behavior.

Condition Action Example

This example shows the behavior of a simple condition action in a multiple segment transition.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Conditions `C_one` and `C_two` are false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment from state `A` to a connective junction is detected. The condition action `A_one` is detected on the valid transition segment and is immediately executed and completed. State `A` is still active.

2. Because the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.

3. State `A` `during` actions (`durA()`) execute and complete.

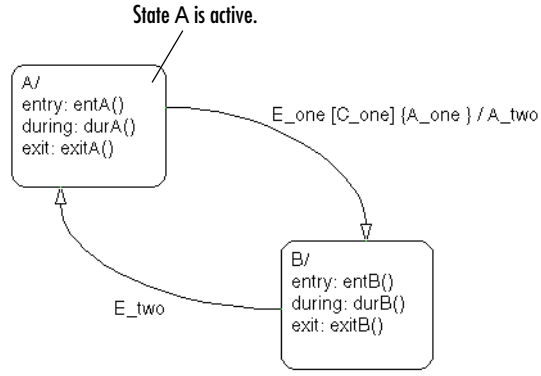
   State `A` remains active.

4. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when state `A` is initially active.

Condition and Transition Actions Example

This example shows the behavior of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.



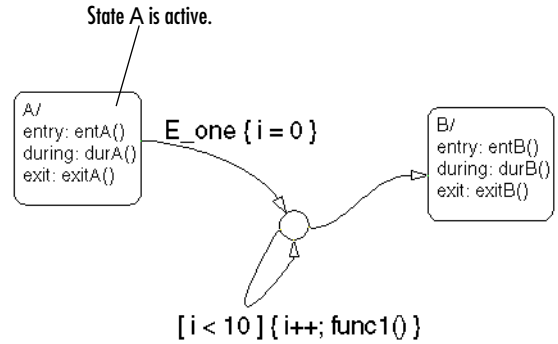
Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `C_one` is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `A` to state `B` is detected. The condition `C_one` is true. The condition action `A_one` is detected on the valid transition and is immediately executed and completed. State `A` is still active.
2. State `A` `exit` actions (`ExitA()`) execute and complete.
3. State `A` is marked inactive.
4. The transition action `A_two` is executed and completed.
5. State `B` is marked active.
6. State `B` `entry` actions (`entB()`) execute and complete.
7. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when state `A` is initially active.

Condition Actions in For Loop Construct Example

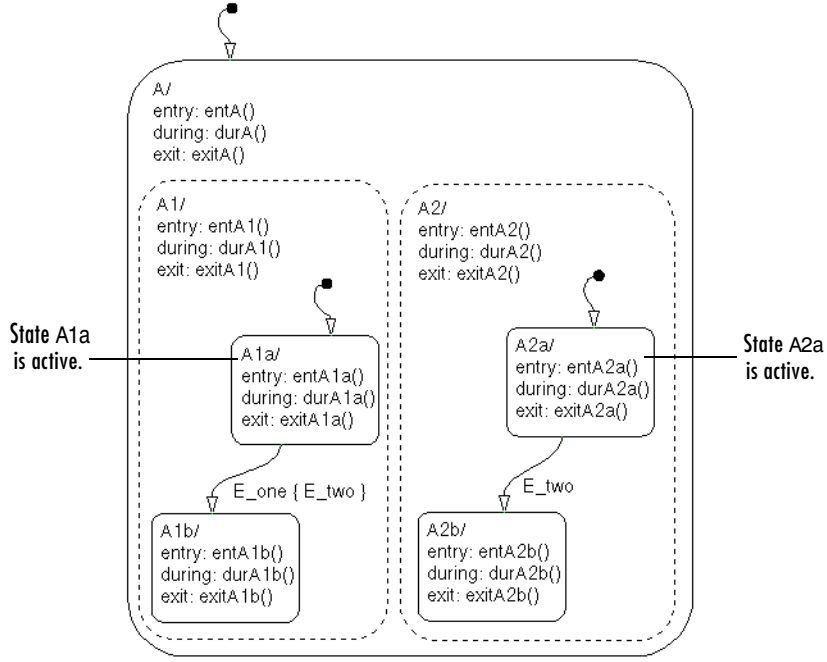
Condition actions and connective junctions are used to design a `for` loop construct. This example shows the use of a condition action and connective junction to create a `for` loop construct.



See “For Loop Construct Example” on page 4-54 to see the behavior of this example.

Condition Actions to Broadcast Events to Parallel (AND) States Example

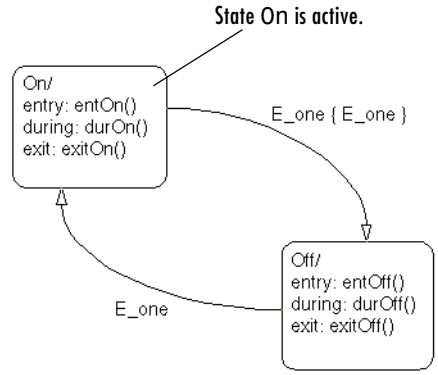
This example shows the use of condition actions used to broadcast events immediately to parallel (AND) states.



See “Event Broadcast Condition Action Example” on page 4-72 to see the behavior of this example.

Cyclic Behavior to Avoid with Condition Actions Example

This example shows a notation to avoid when using event broadcasts as condition actions because the semantics result in cyclic behavior.



Initially the Stateflow diagram is asleep. State `On` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`.

   A valid transition from state `On` to state `Off` is detected.

2. The condition action on the transition broadcasts event `E_one`.

3. Event `E_one` is detected on the valid transition, which is immediately executed. State `On` is still active.

4. The broadcast of event `E_one` awakens the Stateflow diagram a second time.

5. Go to step 1.

Step 1 to 5 continue to execute in a cyclical manner. The transition label indicating a trigger on the same event as the condition action broadcast event results in unrecoverable cyclic behavior. This sequence never completes when event `E_one` is broadcast and state `On` is active.

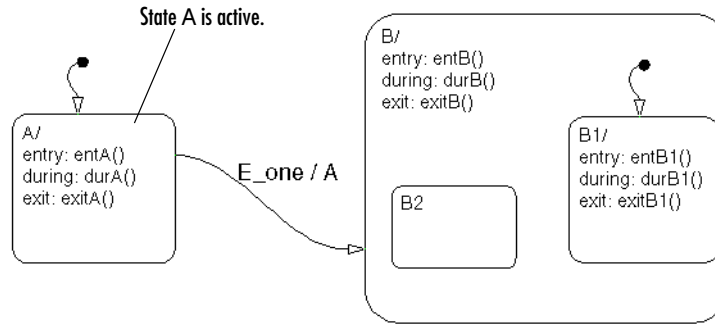
Default Transition Examples

The following examples demonstrate the use of default transitions in Stateflow:

- "Default Transition in Exclusive (OR) Decomposition Example" on page 4-35 — Shows the behavior of a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.
- "Default Transition to a Junction Example" on page 4-36 — Shows the behavior of a default transition to a connective junction.
- "Default Transition and a History Junction Example" on page 4-37 — Shows the behavior of a superstate with a default transition and a history junction.
- "Labeled Default Transitions Example" on page 4-39 — Shows the use of a default transition with a label.

Default Transition in Exclusive (OR) Decomposition Example

This example shows a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.



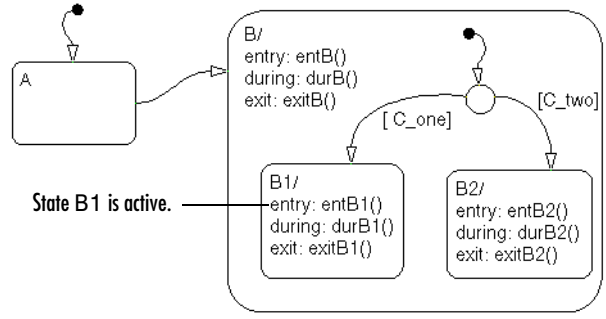
Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition from state `A` to superstate `B`.
2. State `A` `exit` actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. The transition action, `A`, is executed and completed.
5. State `B` is marked active.
6. State `B` `entry` actions (`entB()`) execute and complete.
7. State `B` detects a valid default transition to state `B.B1`.
8. State `B.B1` is marked active.
9. State `B.B1` `entry` actions (`entB1()`) execute and complete.
10. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when state `A` is initially active.

Default Transition to a Junction Example

The following example shows the behavior of a default transition to a connective junction.



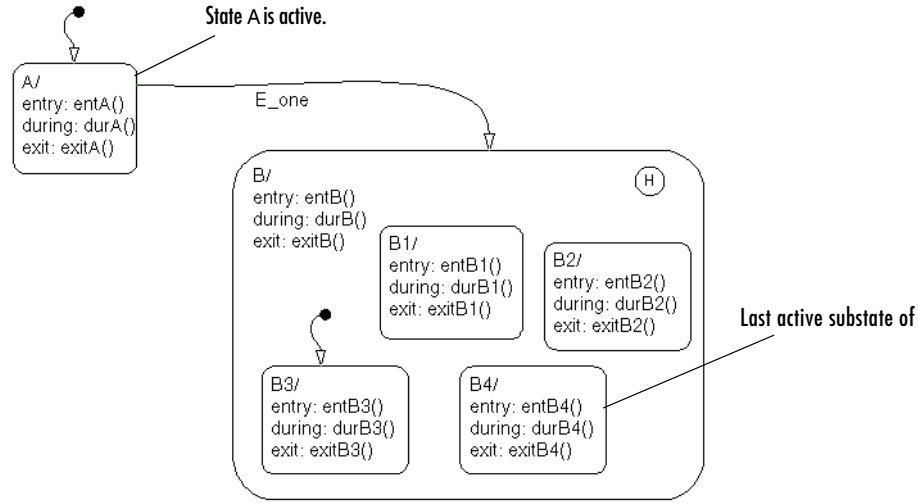
Initially the Stateflow diagram is asleep. State `B.B1` is active. An event occurs and awakens the Stateflow diagram. Condition `[C_two]` is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. State `B` checks to see if there is a valid transition as a result of any event. There is none.
2. State `B1` `during` actions (`durB1()`) execute and complete.

This sequence completes the execution of this Stateflow diagram associated with the occurrence of any event.

Default Transition and a History Junction Example

This example shows the behavior of a superstate with a default transition and a history junction.



Initially the Stateflow diagram is asleep. State `A` is active. There is a history junction and state `B4` was the last active substate of superstate `B`. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`.

   There is a valid transition from state `A` to superstate `B`.

2. State `A` `exit` actions (`exitA()`) execute and complete.

3. State `A` is marked inactive.

4. State `B` is marked active.

5. State `B` `entry` actions (`entB()`) execute and complete.

6. State `B` uses the history junction to determine the substate destination of the transition into the superstate.

   The history junction indicates that substate `B.B4` was the last active substate, which becomes the destination of the transition.

7. State `B.B4` is marked active.

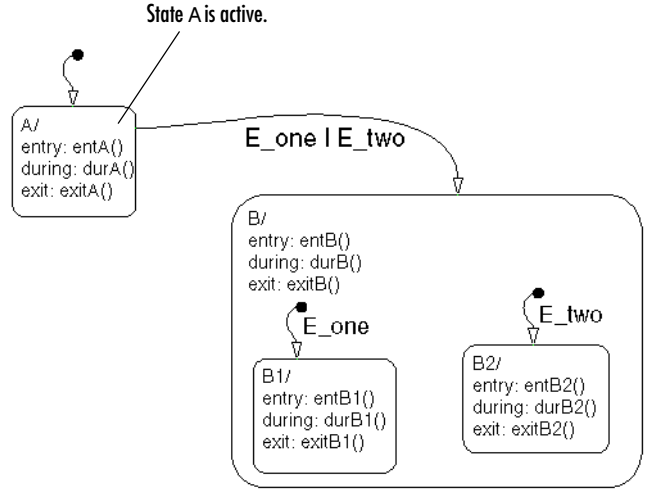
8. State `B.B4` `entry` actions (`entB4()`) execute and complete.

9. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Labeled Default Transitions Example

This example shows the use of a default transition with a label.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs, awakening the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram with the following steps:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`.

   There is a valid transition from state `A` to superstate `B`. The transition is valid if event `E_one` or `E_two` occurs.

2. State `A` `exit` actions execute and complete (`exitA()`).

3. State `A` is marked inactive.

4. State `B` is marked active.

5. State `B` `entry` actions execute and complete (`entB()`).

6. State `B` detects a valid default transition to state `B.B1`. The default transition is valid as a result of `E_one`.

7 State B.B1 is marked active.

8 State B.B1 entry actions execute and complete (entB1()).

9 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when state A is initially active.

Inner Transition Examples

The following examples demonstrate the use of inner transitions in Stateflow:

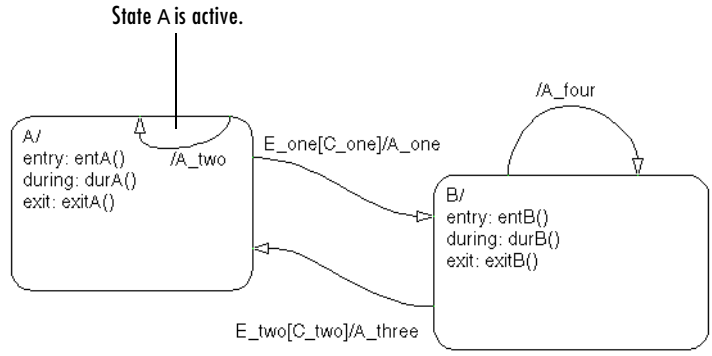
- "Processing Events with an Inner Transition in an Exclusive (OR) State Example" on page 4-41 — Shows what happens when processing repeated events using an inner transition in an exclusive (OR) state.
- "Processing Events with an Inner Transition to a Connective Junction Example" on page 4-45 — Shows the behavior of handling repeated events using an inner transition to a connective junction.
- "Inner Transition to a History Junction Example" on page 4-48 — Shows the behavior of an inner transition to a history junction.

Processing Events with an Inner Transition in an Exclusive (OR) State Example

This example shows what happens when processing three events using an inner transition in an exclusive (OR) state.

Processing One Event in an Exclusive (OR) State

This example shows the behavior of an inner transition.



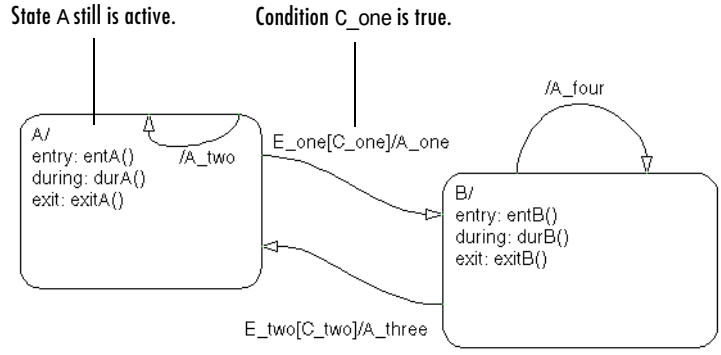
Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_one]` is false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A potentially valid transition from state `A` to state `B` is detected. However, the transition is not valid, because `[C_one]` is false.
2. State `A` `during` actions (`durA()`) execute and complete.
3. State `A` checks its children for a valid transition and detects a valid inner transition.
4. State `A` remains active. The inner transition action `A_two` is executed and completed. Because it is an inner transition, state `A`'s `exit` and `entry` actions are not executed.
5. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Processing a Second Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a second event `E_one` occurs.



Initially the Stateflow diagram is asleep. State `A` is still active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_one]` is true. Event

E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram with the following steps:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one.

   The transition from state A to state B is now valid because [C_one] is true.

2. State A exit actions (exitA()) execute and complete.

3. State A is marked inactive.

4. The transition action A_one is executed and completed.

5. State B is marked active.

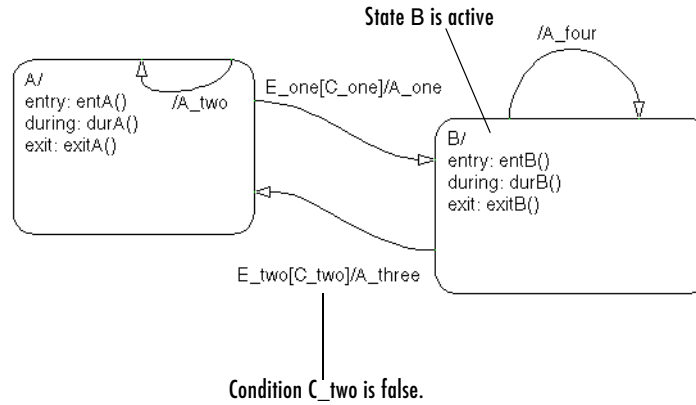
6. State B entry actions (entB()) execute and complete.

7. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Processing a Third Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a third event, E_two, occurs.



Initially the Stateflow diagram is asleep. State `B` is now active. Event `E_two` occurs and awakens the Stateflow diagram. Condition `[C_two]` is false. Event `E_two` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`.

   A potentially valid transition from state `B` to state `A` is detected. The transition is not valid because `[C_two]` is false. However, active state `B` has a valid self-loop transition.

2. State `B` `exit` actions (`exitB()`) execute and complete.

3. State `B` is marked inactive.

4. The self-loop transition action, `A_four`, executes and completes.

5. State `B` is marked active.

6. State `B` `entry` actions (`entB()`) execute and complete.

7. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

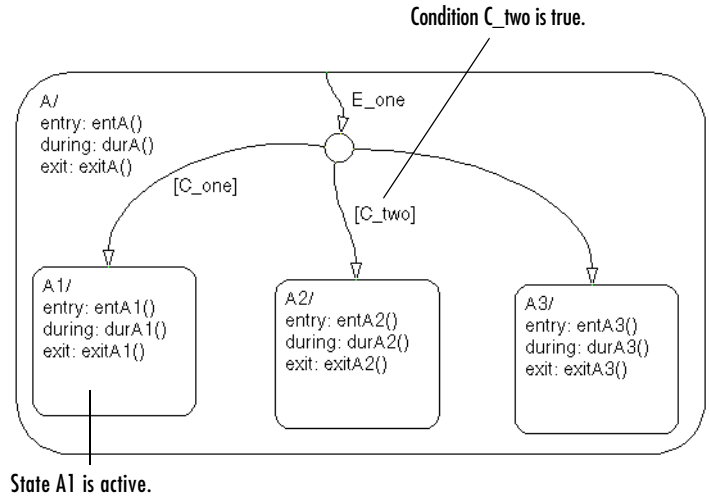
This sequence completes the execution of this Stateflow diagram associated with event `E_two`. This example shows the difference in behavior between inner and self-loop transitions.

Processing Events with an Inner Transition to a Connective Junction Example

This example shows the behavior of handling repeated events using an inner transition to a connective junction.

Processing the First Event with an Inner Transition to a Connective Junction

This example shows the behavior of an inner transition to a connective junction for an initial event.



Initially the Stateflow diagram is asleep. State `A1` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_two]` is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition at the root level as a result of `E_one`. There is no valid transition.

2. State `A` `during` actions (`durA()`) execute and complete.

3. State `A` checks itself for valid transitions and detects that there is a valid inner transition to a connective junction.

   The conditions are evaluated to determine whether one of the transitions is valid. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner. Because `[C_two]` is true, the inner transition to the junction and then to state `A.A2` is valid.

4. State `A.A1` `exit` actions (`exitA1()`) execute and complete.

5. State `A.A1` is marked inactive.

6. State `A.A2` is marked active.

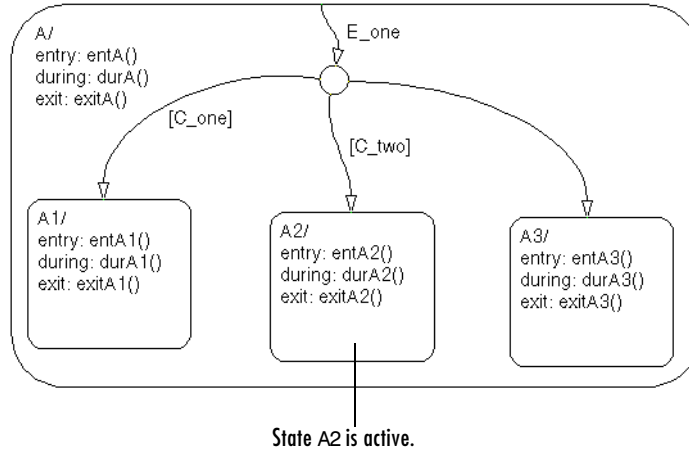
7. State `A.A2` `entry` actions (`entA2()`) execute and complete.

8. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when condition `C_two` is true.

Processing a Second Event with an Inner Transition to a Connective Junction

Continuing the previous example, this example shows the behavior of an inner transition to a junction when a second event `E_one` occurs.



Initially the Stateflow diagram is asleep. State `A2` is active. Event `E_one` occurs and awakens the Stateflow diagram. Neither `[C_one]` nor `[C_two]` is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition at the root level as a result of `E_one`. There is no valid transition.
2. State `A` `during` actions (`durA()`) execute and complete.
3. State `A` checks itself for valid transitions and detects a valid inner transition to a connective junction. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner. Because neither `[C_one]` nor `[C_two]` is true, the unlabeled transition segment is evaluated and is determined to be valid. The full transition from the inner transition to state `A.A3` is valid.
4. State `A.A2` `exit` actions (`exitA2()`) execute and complete.
5. State `A.A2` is marked inactive.
6. State `A.A3` is marked active.

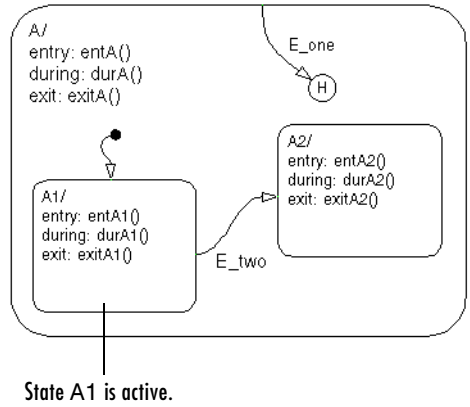
7 State A.A3 entry actions (entA3()) execute and complete.

8 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when neither [C_one] nor [C_two] is true.

Inner Transition to a History Junction Example

This example shows the behavior of an inner transition to a history junction.



Initially the Stateflow diagram is asleep. State A.A1 is active. There is history information because superstate A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.

2 State A during actions execute and complete.

3 State A checks itself for valid transitions and detects that there is a valid inner transition to a history junction. According to the behavior of history junctions, the last active state, A.A1, is the destination state.

4. State `A.A1` `exit` actions execute and complete.
5. State `A.A1` is marked inactive.
6. State `A.A1` is marked active.
7. State `A.A1` `entry` actions execute and complete.
8. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when there is an inner transition to a history junction and state `A.A1` is active.

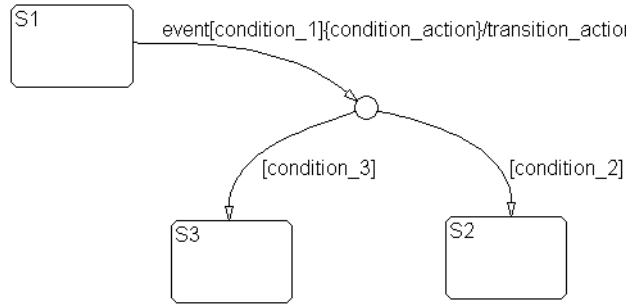
Connective Junction Examples

The following examples demonstrate the use of connective junctions in Stateflow:

- "Label Format for Transition Segments Example" on page 4-50 — Shows the general label format for a transition segment.
- "If-Then-Else Decision Construct Example" on page 4-51 — Shows the behavior of an if-then-else decision construct using a connective junction.
- "Self-Loop Transition Example" on page 4-53 — Shows the behavior of a self-loop transition using a connective junction.
- "For Loop Construct Example" on page 4-54 — Shows the behavior of a for loop using a connective junction.
- "Flow Diagram Notation Example" on page 4-55 — Shows the behavior of a flow notation in a Stateflow diagram.
- "Transitions from a Common Source to Multiple Destinations Example" on page 4-57 — Shows the behavior of transitions from a common source to multiple conditional destinations using a connective junction.
- "Transitions from Multiple Sources to a Common Destination Example" on page 4-59 — Shows the behavior of transitions from multiple sources to a single destination using a connective junction.
- "Transitions from a Source to a Destination Based on a Common Event Example" on page 4-60 — Shows the behavior of transitions from multiple sources to a single destination based on the same event using a connective junction.
- "Backtracking Behavior in Flow Graphs Example" on page 4-61 — Shows the behavior of transitions with junctions that force back-tracking behavior in flow graphs.

Label Format for Transition Segments Example

The general label format for a transition segment entering a junction is the same as for transitions entering states, as shown in the following example:

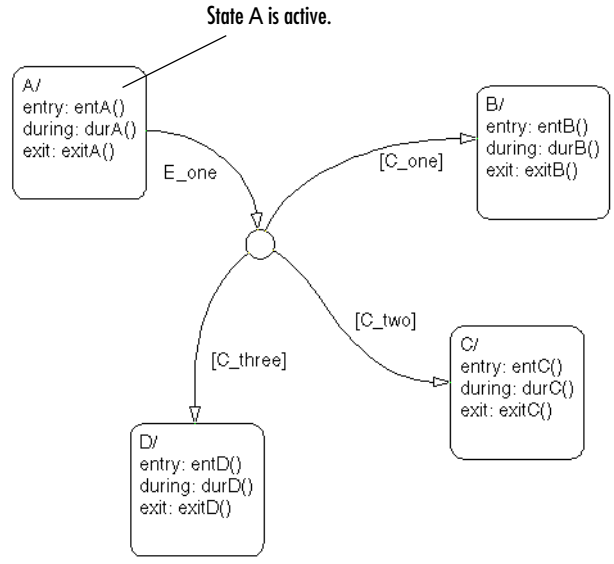


Execution of a transition in this example occurs as follows:

1. When an event occurs, state S1 is checked for an outgoing transition with a matching event specified.
2. If a transition with a matching event is found, the transition condition for that transition (in brackets) is evaluated.
3. If `condition_1` evaluates to true, the condition action `condition_action` (in braces) is executed.
4. The outgoing transitions from the junction are checked for a valid transition. Since `condition_2` is true, a valid state-to-state transition (S1 to S2) is found.
5. State S1 is exited (this includes the execution of S1's `exit` action).
6. The transition action `transition_action` is executed.
7. The completed state-to-state transition (S1 to S2) is taken.
8. State S2 is entered (this includes the execution of S2's `entry` action).

If-Then-Else Decision Construct Example

This example shows the behavior of an `if-then-else` decision construct.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition [`C_two`] is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`.

   There is a valid transition segment from state `A` to the connective junction. The transition segments beginning from a twelve o'clock position on the connective junction are evaluated for validity. The first transition segment, labeled with condition [`C_one`], is not valid. The next transition segment, labeled with the condition [`C_two`], is valid. The complete transition from state `A` to state `C` is valid.

2. State `A` `exit` actions (`exitA()`) execute and complete.

3. State `A` is marked inactive.

4. State `C` is marked active.

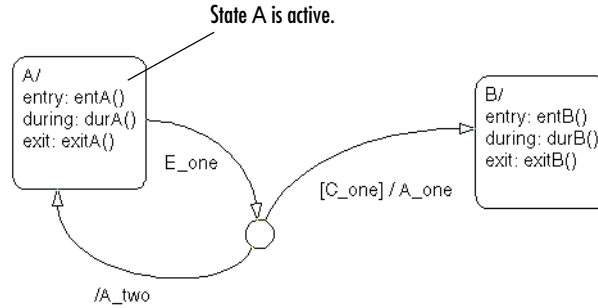
5 State `C entry` actions (`entC()`) execute and complete.

6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Self-Loop Transition Example

This example shows the behavior of a self-loop transition using a connective junction.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition [`C_one`] is false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `A` to the connective junction. The transition segment labeled with a condition and action is evaluated for validity. Because the condition [`C_one`] is not valid, the complete transition from state `A` to state `B` is not valid. The transition segment from the connective junction back to state `A` is valid.

2 State `A exit` actions (`exitA()`) execute and complete.

3 State `A` is marked inactive.

4 The transition action `A_two` is executed and completed.

5 State `A` is marked active.

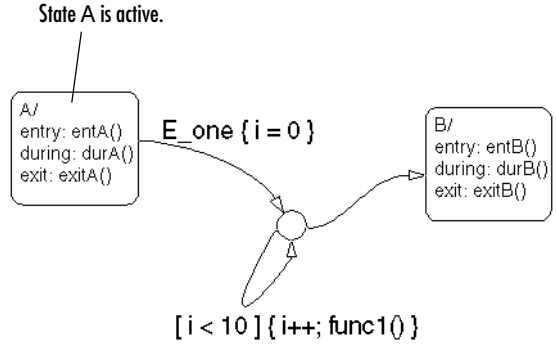
6 State `A` `entry` actions (`entA()`) execute and complete.

7 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

For Loop Construct Example

This example shows the behavior of a `for` loop using a connective junction.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `A` to the connective junction. The transition segment condition action, `i = 0`, is executed and completed. Of the two transition segments leaving the connective junction, the transition segment that is a self-loop back to the connective junction is evaluated next for validity. That segment takes

priority in evaluation because it has a condition specified, whereas the other segment is unlabeled.

2. The condition [`i < 10`] is evaluated as true. The condition actions `i++` and a call to `func1` are executed and completed until the condition becomes false. A connective junction is not a final destination; thus the transition destination remains to be determined.

3. The unconditional segment to state `B` is now valid. The complete transition from state `A` to state `B` is valid.

4. State `A` `exit` actions (`exitA()`) execute and complete.

5. State `A` is marked inactive.

6. State `B` is marked active.

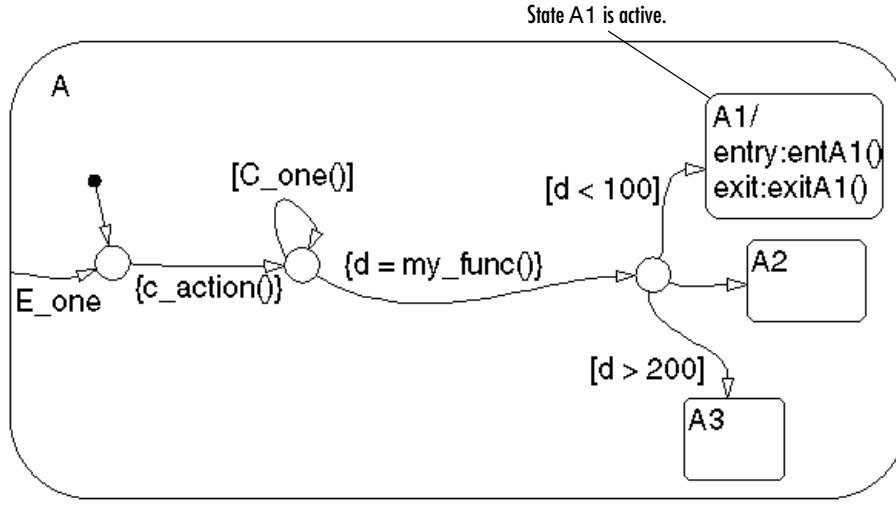
7. State `B` `entry` actions (`entB()`) execute and complete.

8. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Flow Diagram Notation Example

This example shows the behavior of a Stateflow diagram that uses flow notation.



Initially the Stateflow diagram is asleep. State `A.A1` is active. The condition `[C_one()]` is initially true. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2. State `A` checks itself for valid transitions and detects a valid inner transition to a connective junction.

3. The next possible segments of the transition are evaluated. There is only one outgoing transition and it has a condition action defined. The condition action is executed and completed.

4. The next possible segments are evaluated. There are two outgoing transitions; one is a conditional self-loop transition and the other is an unconditional transition segment. The conditional transition segment takes precedence. The condition `[C_one()]` is tested and is true; the self-loop transition is taken. Since a final transition destination has not been reached, this self-loop continues until `[C_one()]` is false.

   Assume that after five iterations `[C_one()]` is false.

5. The next possible transition segment (to the next connective junction) is evaluated. It is an unconditional transition segment with a condition action. The transition segment is taken and the condition action, `{d=my_func()}`, is executed and completed. The returned value of `d` is 84.

6. The next possible transition segment is evaluated. There are three possible outgoing transition segments to consider. Two are conditional; one is unconditional. The segment labeled with the condition `[d<100]` is evaluated first based on the geometry of the two outgoing conditional transition segments. Because the return value of `d` is 84, the condition `[d<100]` is true and this transition (to the destination state `A.A1`) is valid.

7. State `A.A1` exit actions (`exitA1()`) execute and complete.

8. State `A.A1` is marked inactive.

9. State `A.A1` is marked active.

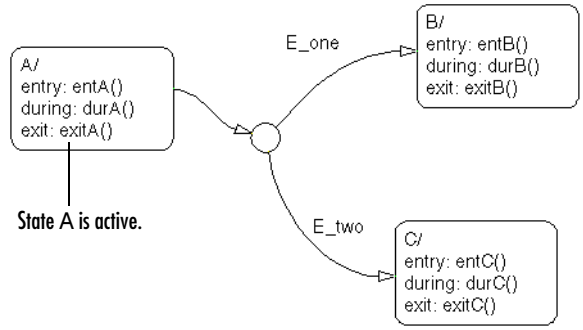
10. State `A.A1` `entry` actions (`entA1()`) execute and complete.

11. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Transitions from a Common Source to Multiple Destinations Example

This example shows the behavior of transitions from a common source to multiple conditional destinations using a connective junction.



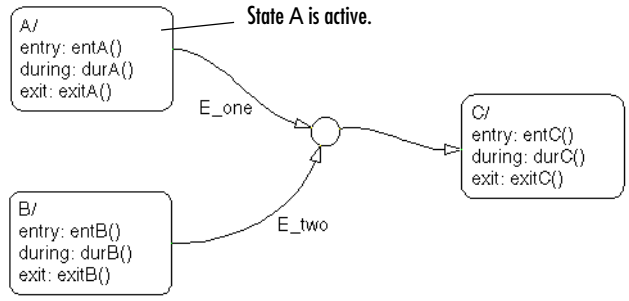
Initially the Stateflow diagram is asleep. State `A` is active. Event `E_two` occurs and awakens the Stateflow diagram. Event `E_two` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is a valid transition segment from state `A` to the connective junction. Given that the transition segments are equivalently labeled, evaluation begins from a twelve o'clock position on the connective junction and progresses clockwise. The first transition segment, labeled with event `E_one`, is not valid. The next transition segment, labeled with event `E_two`, is valid. The complete transition from state `A` to state `C` is valid.
2. State `A` `exit` actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. State `C` is marked active.
5. State `C` `entry` actions (`entC()`) execute and complete.
6. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_two`.

Transitions from Multiple Sources to a Common Destination Example

This example shows the behavior of transitions from multiple sources to a single destination using a connective junction.



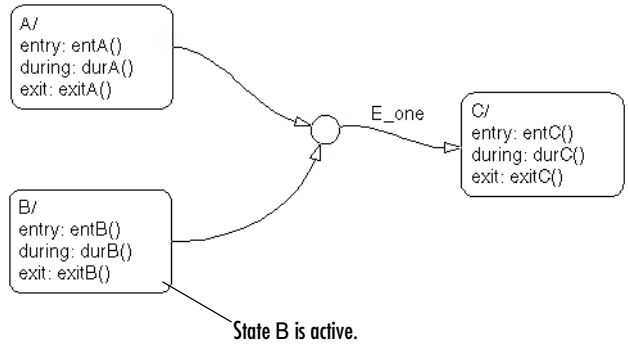
Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `A` to the connective junction and from the junction to state `C`.
2. State `A` `exit` actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. State `C` is marked active.
5. State `C` `entry` actions (`entC()`) execute and complete.
6. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Transitions from a Source to a Destination Based on a Common Event Example

This example shows the behavior of transitions from multiple sources to a single destination based on the same event using a connective junction.



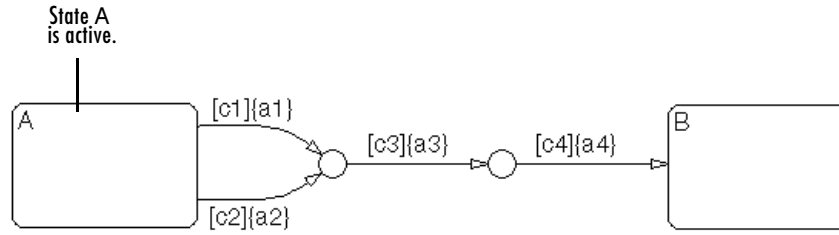
Initially the Stateflow diagram is asleep. State `B` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `B` to the connective junction and from the junction to state C.
2. State `B exit` actions (`exitB()`) execute and complete.
3. State `B` is marked inactive.
4. State `C` is marked active.
5. State `C entry` actions (`entC()`) execute and complete.
6. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Backtracking Behavior in Flow Graphs Example

This example shows the behavior of transitions with junctions that force back-tracking behavior in flow graphs.



Initially, state `A` is active and conditions `c1`, `c2`, and `c3` are true:

1. The Stateflow diagram root checks to see if there is a valid transition from state `A`.

   There is a valid transition segment marked with the condition `c1` from state `A` to a connective junction.

2. Condition `c1` is true, therefore action `a1` is executed.

3. Condition `c3` is true, therefore action `a3` is executed.

4. Condition `c4` is not true, therefore control flow is back-tracked to state `A`.

5. The Stateflow diagram root checks to see if there is another valid transition from state `A`.

   There is a valid transition segment marked with the condition `c2` from state `A` to a connective junction.

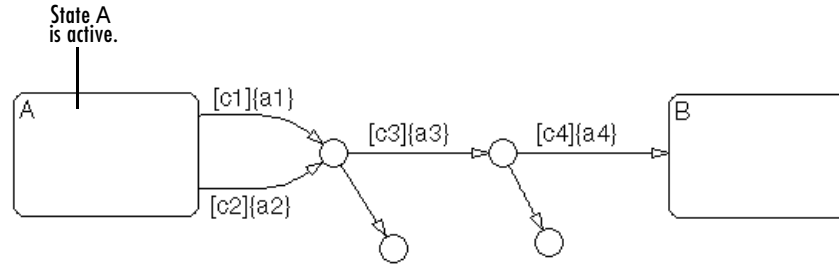
6. Condition `c2` is true, therefore action `a2` is executed.

7. Condition `c3` is true, therefore action `a3` is executed.

8. Condition `c4` is not true, therefore control flow is back-tracked to state `A`.

9. The Stateflow chart goes to sleep.

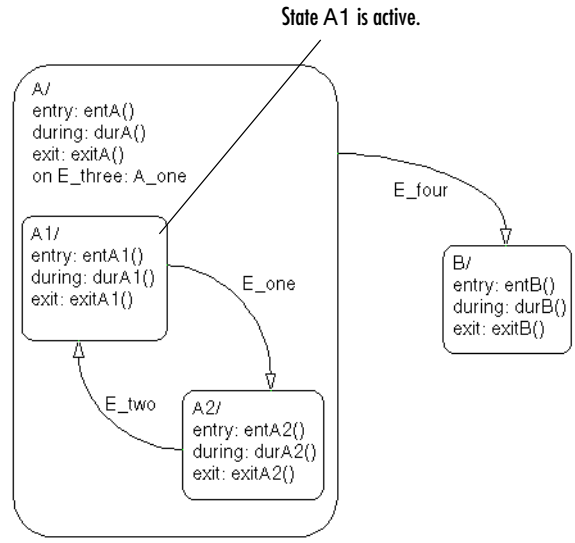
The preceding example shows the unanticipated behavior of executing both actions `a1` and `a2` and executing action `a3` twice. To resolve this problem, consider the following.



In this example, the previous example is amended with two terminating junctions that allow flow to terminate if either `c3` or `c4` is not true. This leaves state A active without taking any unnecessary actions.

Event Actions in a Superstate Example

The following example demonstrates the use of event actions in a superstate:



Initially the Stateflow diagram is asleep. State `A.A1` is active. Event `E_three` occurs and awakens the Stateflow diagram. Event `E_three` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_three`. There is no valid transition.
2. State `A` `during` actions (`durA()`) execute and complete.
3. State `A` executes and completes the `on` event `E_three` action (`A_one`).
4. State `A` checks its children for valid transitions. There are no valid transitions.
5. State `A1` `during` actions (`durA1()`) execute and complete.

6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_three`.

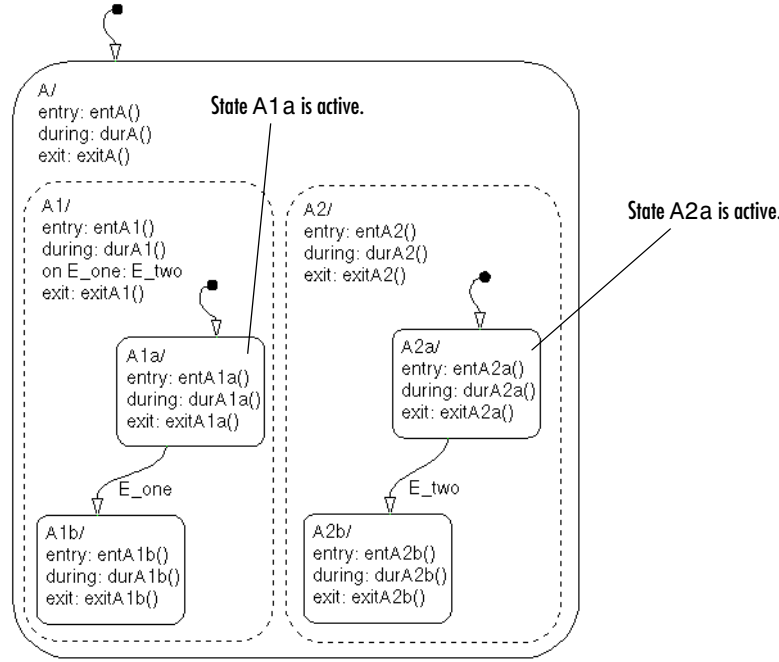
Parallel (AND) State Examples

The following examples demonstrate the use of parallel (AND) states:

- "Event Broadcast State Action Example" on page 4-65 — Shows the behavior of event broadcast actions in parallel states.
- "Event Broadcast Transition Action with a Nested Event Broadcast Example" on page 4-69 — Shows the behavior of an event broadcast transition action that includes a nested event broadcast in a parallel state.
- "Event Broadcast Condition Action Example" on page 4-72 — Shows the behavior of a condition action event broadcast in a parallel (AND) state.

Event Broadcast State Action Example

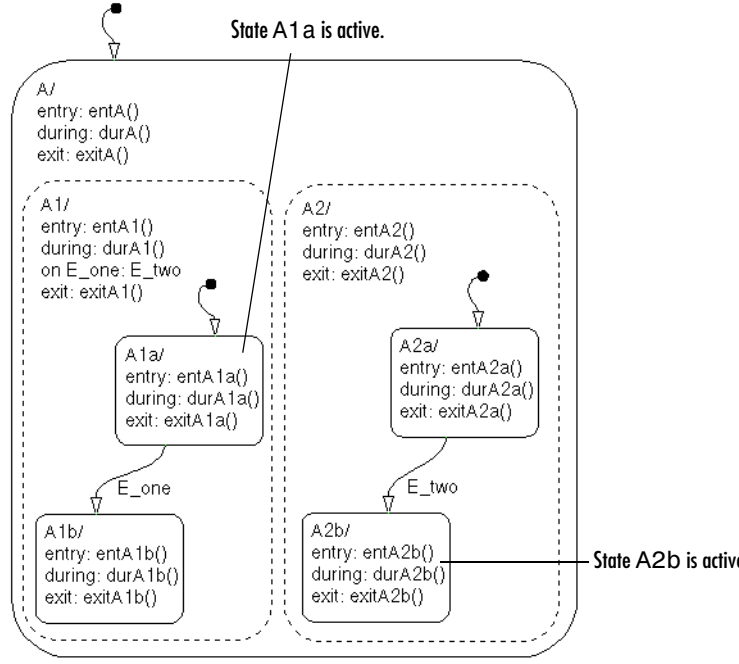
This example shows the behavior of event broadcast actions in parallel states.



Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition at the root level as a result of E_one. There is no valid transition.
2. State A during actions (durA()) execute and complete.
3. State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 during actions (durA1()) execute and complete. State A.A1 executes and completes the on E_one action and broadcasts event E_two. during and on *event_name* actions are processed based on their order of appearance in the state label:
   a. The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.
   b. State A during actions (durA()) execute and complete.
   c. State A checks its children for valid transitions. There are no valid transitions.
   d. State A' s children are evaluated starting with state A.A1. State A.A1 during actions (durA1()) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.
   e. State A.A2 is evaluated. State A.A2 during actions (durA2()) execute and complete. State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.
   f. State A.A2.A2a exit actions (exitA2a()) execute and complete.
   g. State A.A2.A2a is marked inactive.
   h. State A.A2.A2b is marked active.
   i. State A.A2.A2b entry actions (entA2b()) execute and complete. The Stateflow diagram activity now looks like this:

.



4 State A.A1.A1a executes and completes `exit` actions (`exitA1a`).

5 The processing of `E_one` continues once the `on` event broadcast of `E_two` has been processed. State A.A1 checks for any valid transitions as a result of event `E_one`. There is a valid transition from state A.A1.A1a to state A.A1.A1b.

6 State A.A1.A1a is marked inactive.

7 State A.A1.A1b `entry` actions (`entA1b()`) execute and complete.

8 State A.A1.A1b is marked active.

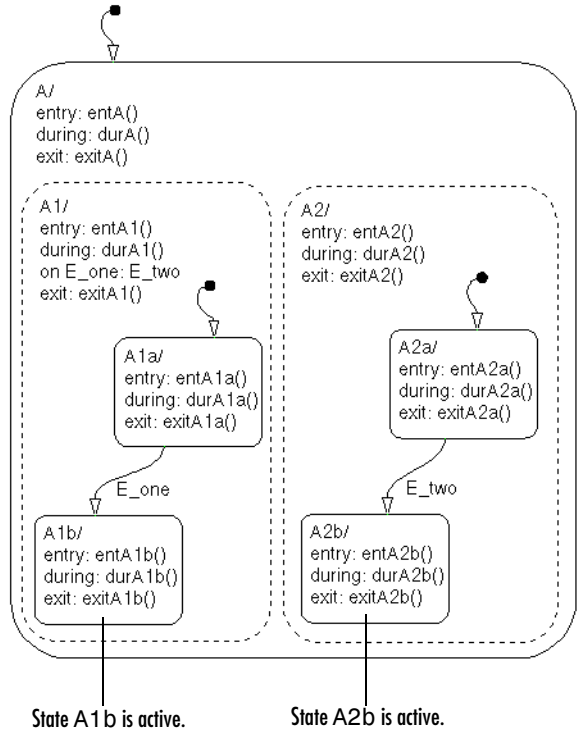
9 Parallel state A.A2 is evaluated next. State A.A2 `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

10 State `A.A2.A2b` `during` actions (`durA2b()`) execute and complete.

State `A.A2.A2b` is now active as a result of the processing of the `on` event broadcast of `E_two`.

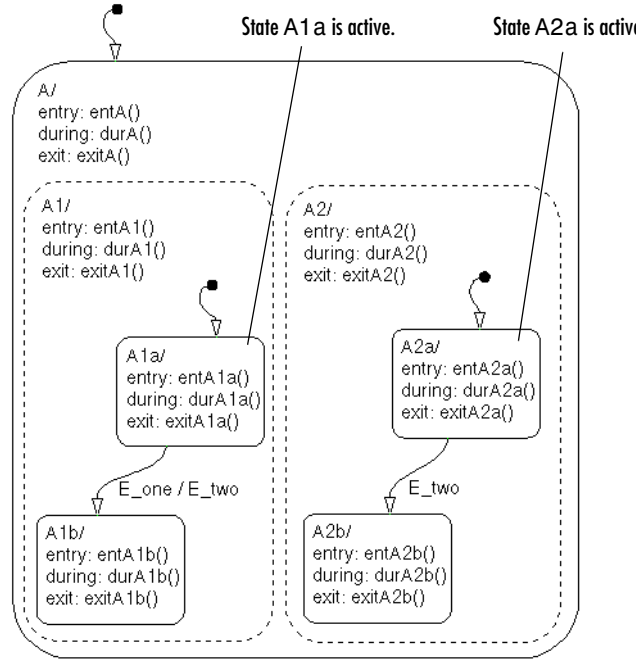
11 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` and the `on` event broadcast to a parallel state of event `E_two`. The final Stateflow diagram activity looks like this.



Event Broadcast Transition Action with a Nested Event Broadcast Example

This example shows the behavior of an event broadcast transition action that includes a nested event broadcast in a parallel state.



Start of Event E_one Processing

Initially the Stateflow diagram is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2. State `A` `during` actions (`durA()`) execute and complete.

3. State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 during actions (durA1()) execute and complete.

4. State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.

5. State A.A1.A1a executes and completes exit actions (exitA1a).

6. State A.A1.A1a is marked inactive.

Event E_two Preempts E_one

7. Transition action generating event E_two is executed and completed:

   a. The transition from state A1a to state A1b (as a result of event E_one) is now preempted by the broadcast of event E_two.

   b. The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.

   c. State A during actions (durA()) execute and complete.

   d. State A' s children are evaluated starting with state A.A1. State A.A1 during actions (durA1()) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.

   e. State A.A2 is evaluated. State A.A2 during actions (durA2()) execute and complete. State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.

   f. State A.A2.A2a exit actions (exitA2a()) execute and complete.

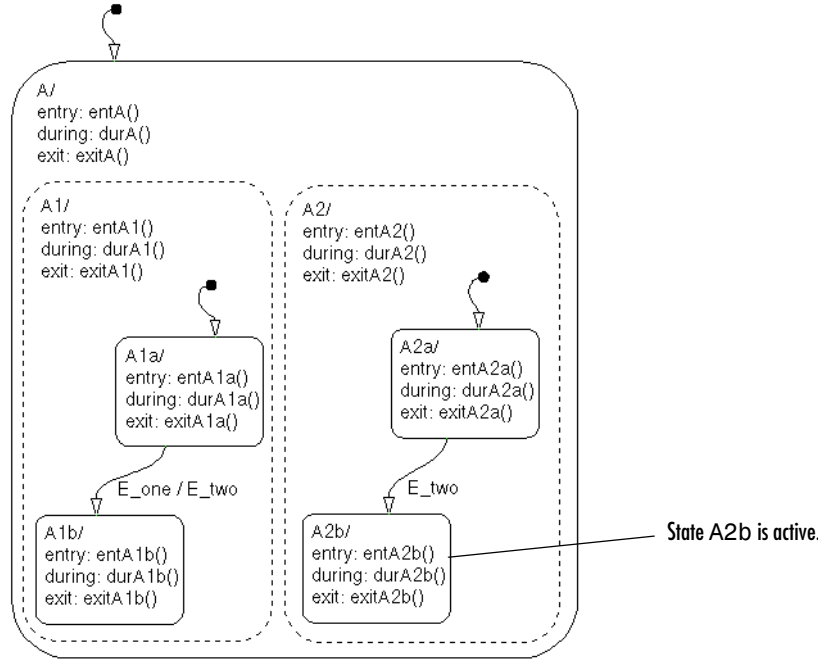
   g. State A.A2.A2a is marked inactive.

   h. State A.A2.A2b is marked active.

   i. State A.A2.A2b entry actions (entA2b()) execute and complete.

Event E_two Processing Ends

The Stateflow diagram activity now looks like this.



Event E_one Processing Resumes

8 State `A.A1.A1b` is marked active.

9 State `A.A1.A1b` `entry` actions (`entA1b()`) execute and complete.

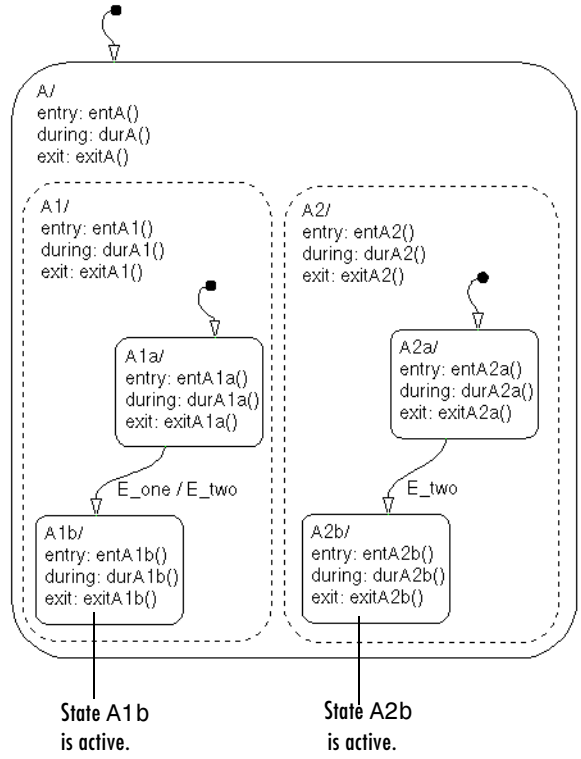
10 Parallel state `A.A2` is evaluated next. State `A.A2` `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

11 State `A.A2.A2b` `during` actions (`durA2b()`) execute and complete.

State `A.A2.A2b` is now active as a result of the processing of the transition action event broadcast of `E_two`.

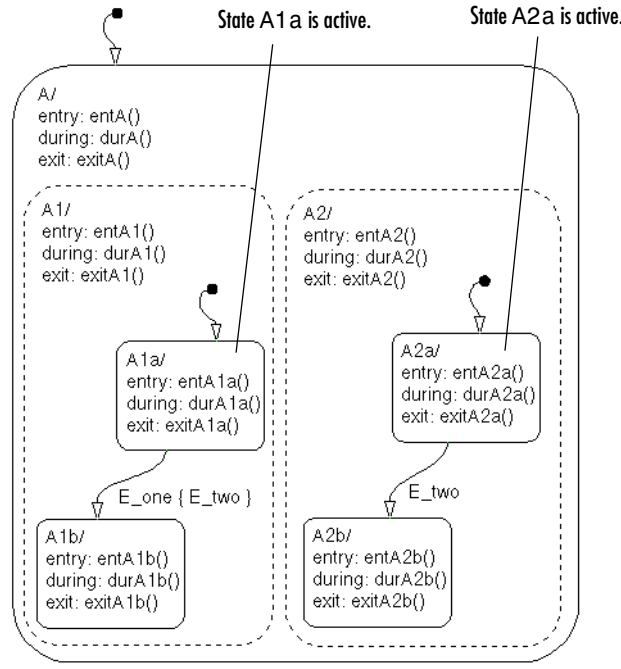
12 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` and the transition action event broadcast to a parallel state of event `E_two`. The final Stateflow diagram activity now looks like this.



Event Broadcast Condition Action Example

This example shows the behavior of a condition action event broadcast in a parallel (AND) state.



Initially the Stateflow diagram is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2. State `A` `during` actions (`durA()`) execute and complete.

3. State A's children are parallel (AND) states. Parallel states are evaluated and executed from top to bottom. In the case of a tie, they are evaluated from left to right. State `A.A1` is evaluated first. State `A.A1` `during` actions (`durA1()`) execute and complete.

4. State `A.A1` checks for any valid transitions as a result of event `E_one`. There is a valid transition from state `A.A1.A1a` to state `A.A1.A1b`. There is also a

valid condition action. The condition action event broadcast of `E_two` is executed and completed. State `A.A1.A1a` is still active:

a The broadcast of event `E_two` awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is no valid transition.

b State A `during` actions (`durA()`) execute and complete.

c State A' s children are evaluated starting with state `A.A1`. State `A.A1` `during` actions (`durA1()`) execute and complete. State `A.A1` is evaluated for valid transitions. There are no valid transitions as a result of `E_two` within state `A1`.

d State `A.A2` is evaluated. State `A.A2` `during` actions (`durA2()`) execute and complete. State `A.A2` checks for valid transitions. State `A.A2` has a valid transition as a result of `E_two` from state `A.A2.A2a` to state `A.A2.A2b`.

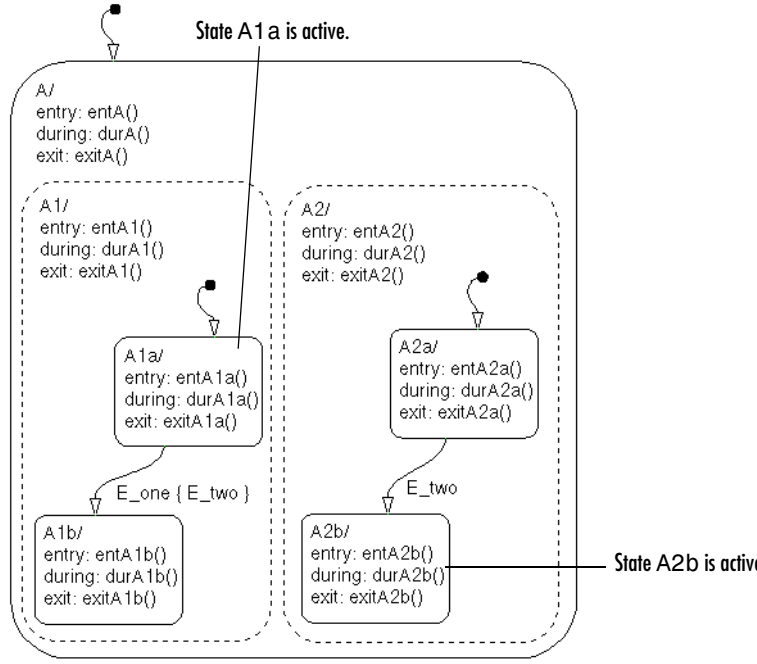
e State `A.A2.A2a` `exit` actions (`exitA2a()`) execute and complete.

f State `A.A2.A2a` is marked inactive.

g State `A.A2.A2b` is marked active.

h State `A.A2.A2b` `entry` actions (`entA2b()`) execute and complete.

The Stateflow diagram activity now looks like this.



5 State A.A1.A1a executes and completes `exit` actions (`exitA1a`).

6 State A.A1.A1a is marked inactive.

7 State A.A1.A1b `entry` actions (`entA1b()`) execute and complete.

8 State A.A1.A1b is marked active.

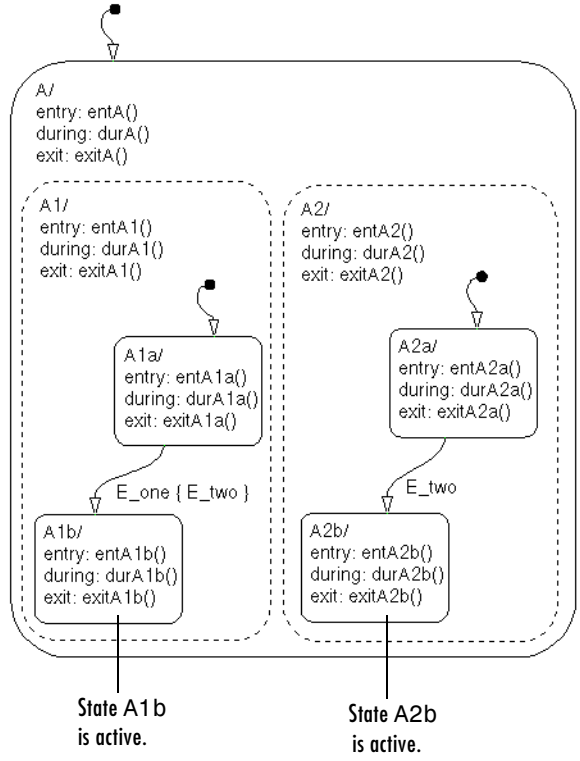
9 Parallel state A.A2 is evaluated next. State A.A2 `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

10 State A.A2.A2b `during` actions (`durA2b()`) execute and complete.

State A.A2.A2b is now active as a result of the processing of the condition action event broadcast of `E_two`.

11 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` and the condition action event broadcast to a parallel state of event `E_two`. The final Stateflow diagram activity now looks like this.



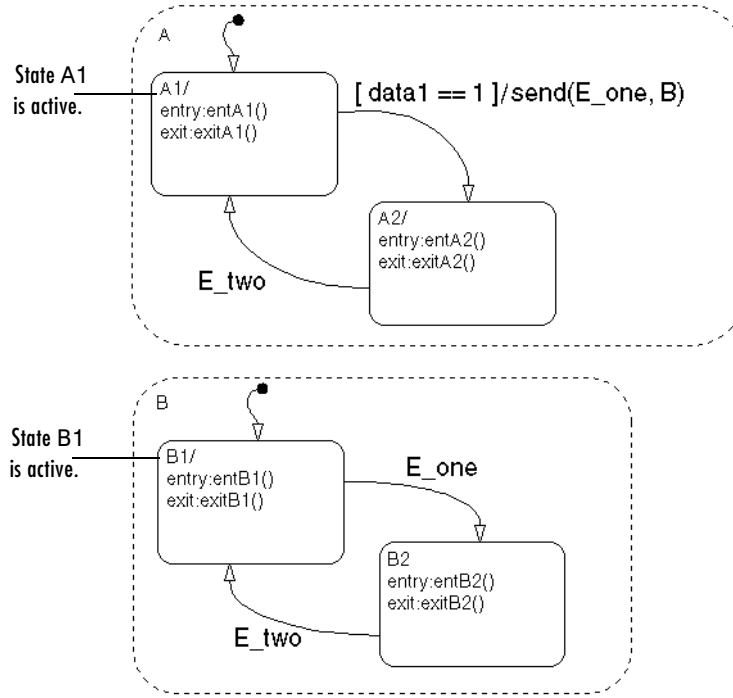
Directed Event Broadcasting Examples

The following examples demonstrate the use of directed event broadcasting:

- "Directed Event Broadcast Using send Example" on page 4-77 — Shows the behavior of directed event broadcast using the send function in a transition action
- "Directed Event Broadcasting Using Qualified Event Names Example" on page 4-79 — Shows the behavior of directed event broadcast using a qualified event name in a transition action

Directed Event Broadcast Using send Example

This example shows the behavior of directed event broadcast using the `send(event_name,state_name)` function in a transition action.



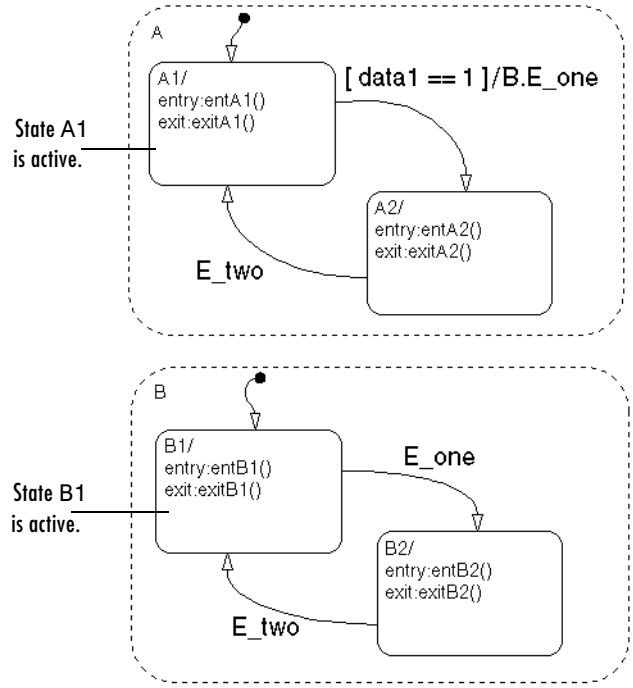
Initially the Stateflow diagram is asleep. Parallel substates `A.A1` and `B.B1` are active. By definition, this implies that parallel (AND) superstates `A` and `B` are active. An event occurs and awakens the Stateflow diagram. The condition `[data1==1]` is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1. The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.
2. State `A` checks for any valid transitions as a result of the event. Because the condition `[data1==1]` is true, there is a valid transition from state `A.A1` to state `A.A2`.
3. State `A.A1` exit actions (`exitA1()`) execute and complete.
4. State `A.A1` is marked inactive.
5. The transition action `send(E_one,B)` is executed and completed:
   a. The broadcast of event `E_one` awakens state `B`. (This is a nested event broadcast.) Because state `B` is active, the directed broadcast is received and state `B` checks to see if there is a valid transition. There is a valid transition from `B.B1` to `B.B2`.
   b. State `B.B1` `exit` actions (`exitB1()`) execute and complete.
   c. State `B.B1` is marked inactive.
   d. State `B.B2` is marked active.
   e. State `B.B2` entry actions `(entB2())` execute and complete.
6. State `A.A2` is marked active.
7. State `A.A2` entry actions (`entA2()`) execute and complete.

This sequence completes the execution of this Stateflow diagram associated with an event broadcast and the directed event broadcast to a parallel state of event `E_one`.

Directed Event Broadcasting Using Qualified Event Names Example

This example shows the behavior of directed event broadcast using a qualified event name in a transition action.



Initially the Stateflow diagram is asleep. Parallel substates `A.A1` and `B.B1` are active. By definition, this implies that parallel (AND) superstates `A` and `B` are active. An event occurs and awakens the Stateflow diagram. The condition `[data1==1]` is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

1 The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.

2. State A checks for any valid transitions as a result of the event. Because the condition [data1==1] is true, there is a valid transition from state A.A1 to state A.A2.

3. State A.A1 exit actions (exitA1()) execute and complete.

4. State A.A1 is marked inactive.

5. The transition action, a qualified event broadcast of event E_one to state B (represented by the notation B.E_one), is executed and completed:

   a. The broadcast of event E_one awakens state B. (This is a nested event broadcast.) Because state B is active, the directed broadcast is received and state B checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.

   b. State B.B1 exit actions (exitB1()) execute and complete.

   c. State B.B1 is marked inactive.

   d. State B.B2 is marked active.

   e. State B.B2 entry actions (entB2()) execute and complete.

6. State A.A2 is marked active.

7. State A.A2 entry actions (entA2()) execute and complete.

This sequence completes the execution of this Stateflow diagram associated with an event broadcast using a qualified event name to a parallel state.

5

Working with Charts

This chapter takes you through the steps of creating graphical Stateflow objects in the Stateflow diagram editor. It includes the following sections:

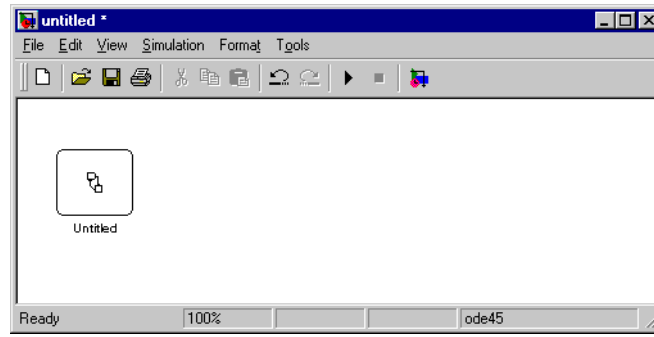
Creating a Stateflow Chart (p. 5-3)	Gives a step-by-step procedure for creating an empty Stateflow chart.
Using the Stateflow Editor (p. 5-6)	Describes each part of the Stateflow diagram editor window that displays the chart you create.
Using States in Stateflow Charts (p. 5-18)	Describes how to create and specify a state in your new chart. Stateflow diagrams react to events by changing states, which are modes of a chart.
Using Transitions in Stateflow Charts (p. 5-28)	Describes how to create, move, change, and specify properties for Stateflow transitions. Charts change active states using pathways called transitions.
Using Boxes in Stateflow Charts (p. 5-47)	Describes Stateflow boxes and how to create them in your new chart. Boxes are a convenience for grouping items in your charts.
Using Graphical Functions in Stateflow Charts (p. 5-48)	Describes how Stateflow graphical functions are created, called, and made available to Simulink. Graphical functions are a convenience, as functions are to programs.
Using Junctions in Stateflow Charts (p. 5-57)	Describes how to create, move, and specify properties for Stateflow junctions. Junctions provide decision points between alternate transition paths. History junctions record the activity of states inside states.
Using Notes in Stateflow Charts (p. 5-61)	Shows you how to create, edit, and delete descriptive notes for your Stateflow chart.

Using Subcharts in Stateflow Charts (p. 5-64)	Shows you how to create and work with charts within charts, that is, subcharts. Subcharts are a convenience for compacting your diagrams.
Using Supertransitions in Stateflow Charts (p. 5-71)	Shows you how to make a supertransition to connect transitions from outside a subchart to a state or junction inside a subchart.
Specifying Chart Properties (p. 5-78)	Tells you how to specify properties for your chart. Part of a chart's interface to its Simulink model is set when you specify the properties for a chart.
Checking the Chart for Errors (p. 5-83)	Shows you how Stateflow parses the diagram to check for errors in diagrams that you create.
Creating Chart Libraries (p. 5-84)	Shows you how to save Stateflow charts that you can place in the Simulink block library for repeated use in a Simulink model.
Printing and Reporting on Charts (p. 5-85)	Shows you the options Stateflow offers for printing part or all of your Stateflow chart.

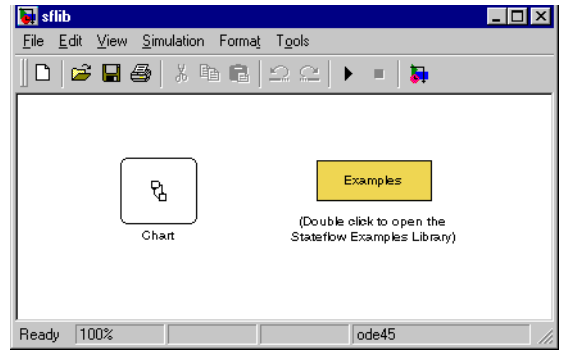
Creating a Stateflow Chart

Charts contain a Stateflow diagram that you build with Stateflow objects. You create charts by adding them to a Simulink system. Create a Stateflow chart in a Simulink system with the following steps:

1 Enter `sfnew` or `stateflow` at the MATLAB command prompt to create a new empty model with a Stateflow chart.



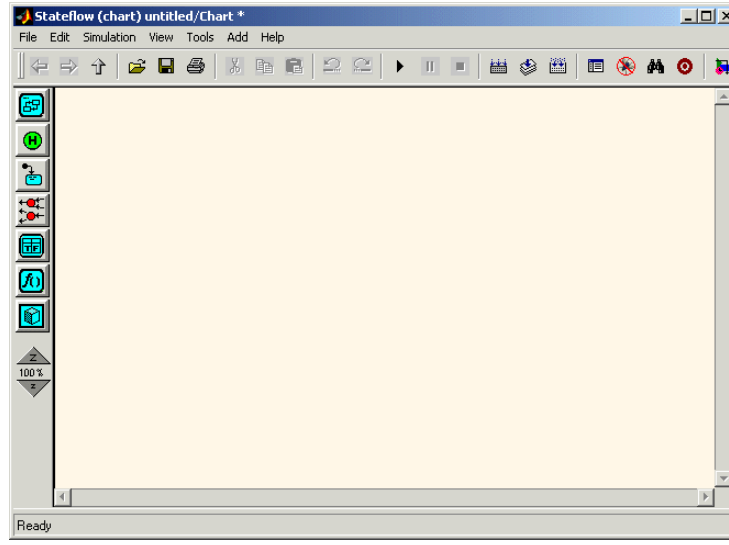
The `stateflow` command also displays the Stateflow block library.



You can drag and drop additional charts in your Simulink system from this library in case you want to create multiple charts in your model. You can also drag and drop new charts into existing systems from the Stateflow library in the Simulink Library browser. For information on creating your own chart libraries, see “Creating Chart Libraries” on page 5-84.

2 Open the chart by double-clicking the Chart block.

Stateflow opens the empty chart in a Stateflow editor window.



3 Use the Stateflow editor to draw a Stateflow chart diagram.

See “Using the Stateflow Editor” on page 5-6 and the remaining sections in this chapter for more information on how to draw Stateflow diagrams.

Once you have constructed a Stateflow diagram, continue with the remaining steps to integrate your chart into its Simulink model:

4 Specify an update method for the chart.

You set the update method for a chart by setting the **Update Method** field in the properties dialog for the chart. This value determines when and how often the chart is called during the execution of the Simulink model that calls this chart. See “Specifying Chart Properties” on page 5-78.

5 Interface the chart to other blocks in your Stateflow model, using events and data.

   See “Defining Events and Data” on page 6-1 and “Defining Stateflow Interfaces” on page 8-1 for more information.

6 Rename and save the model chart by selecting **Save Model As** from the Stateflow editor menu or **Save As** from the Simulink menu.

Note Trying to save a model with more than 25 characters produces an error. Loading a model with more than 25 characters produces a warning.

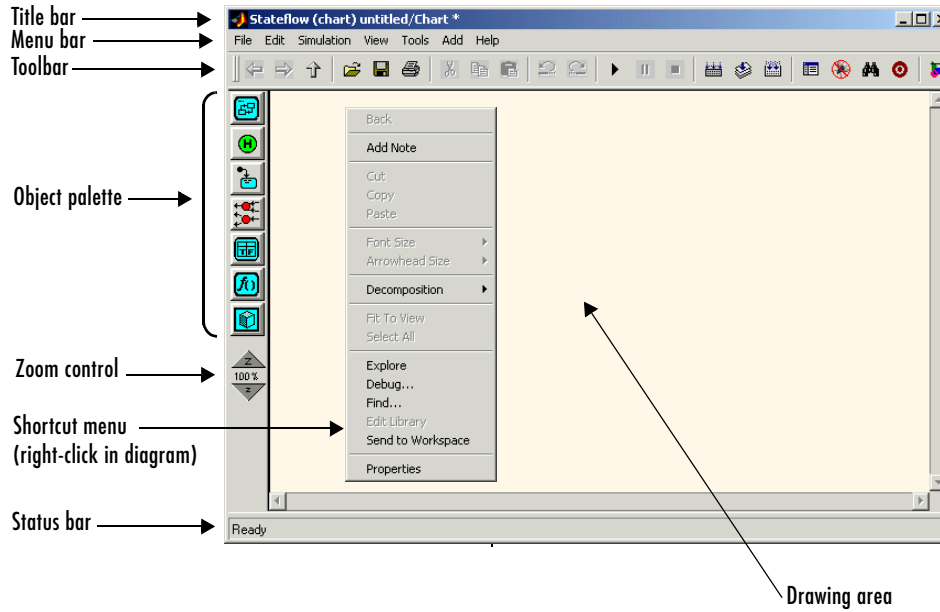
Using the Stateflow Editor

You edit your Stateflow chart diagrams in the Stateflow Editor. This section describes each part of the Stateflow diagram editor window displaying the chart you created. It contains the following topics:

- "Stateflow Diagram Editor Window" on page 5-6 — Describes the parts of the window in which you edit Stateflow diagrams.
- "Drawing Objects" on page 5-8 — Teaches you how to draw the graphical objects in a Stateflow diagram.
- "Displaying the Context Menu for Objects" on page 5-10 — Shows you how to use the right-click context menu for doing operations on that object.
- "Specifying Colors and Fonts" on page 5-10 — Shows you how to change the colors and fonts for all the graphical objects in a Stateflow diagram.
- "Selecting and Deselecting Objects" on page 5-13 — Shows you how to select or deselect objects in the diagram editor to edit them.
- "Cutting and Pasting Objects" on page 5-13 — Tells you how to cut and paste objects in a Stateflow diagram from one location to another.
- "Copying Objects" on page 5-14 — Shows you how to copy objects in the diagram editor.
- "Editing Object Labels" on page 5-14 — Teaches you how to edit the labels of state and transition objects in the diagram editor.
- "Viewing Data and Events from the Editor" on page 5-15 — Use the Stateflow Explorer to add, change, or view data and events for a Stateflow diagram.
- "Zooming a Diagram" on page 5-15 — Shows you how to zoom in to get a closer look at parts of your Stateflow diagram.
- "Undoing and Redoing Editor Operations" on page 5-16 — Shows you how to undo and redo operations you perform in the Stateflow diagram editor.

Stateflow Diagram Editor Window

You use the Stateflow diagram editor to draw, zoom, modify, print, and save a state diagram displayed in the window. It has the following appearance:



The Stateflow diagram editor window includes the following elements:

- Title bar

  The full chart name appears here in *model name*/*chart name** format. The * character appears on the end of the chart name for a newly created chart or for an existing chart that has been edited but not saved yet.

- Menu bar

  Most editor commands are available from the menu bar.

- Toolbar

  Contains buttons for cut, copy, paste, and other commonly used editor commands. You can identify each tool of the toolbar by placing the mouse cursor over it until an identifying tool tip appears.

  The toolbar also contains buttons for navigating a chart's subchart hierarchy (see "Navigating Subcharts" on page 5-69).

- Object palette

  Displays a set of tools for drawing states, transitions, and other state chart objects. See “Drawing Objects” on page 5-8 for more information.

- Drawing area

  Displays an editable copy of a state diagram.

- Zoom control

  See “Viewing Data and Events from the Editor” on page 5-15 for information on using the zoom control.

- Shortcut menus

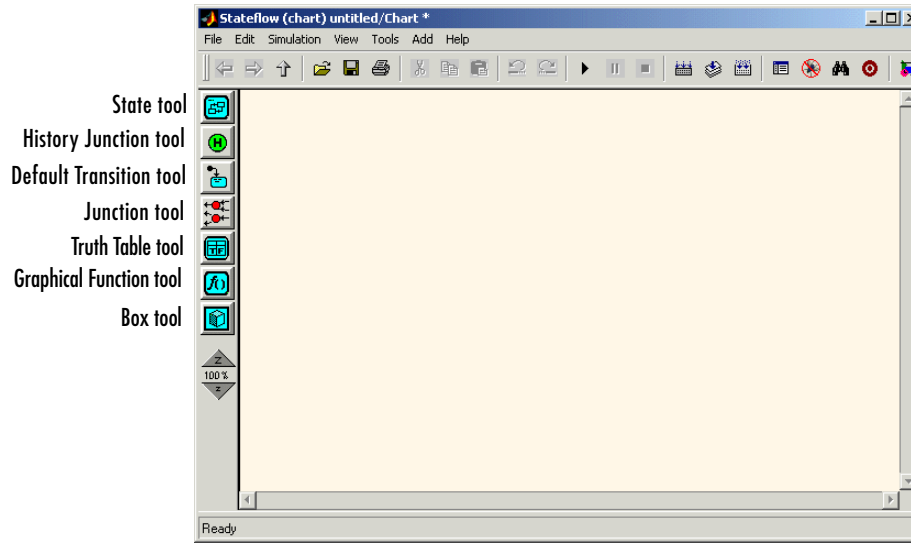
  These menus pop up from the drawing area when you right-click an object. They display commands that apply only to that object. If you right-click an empty area of the diagram editor, the shortcut menu applies to the chart object. See “Displaying the Context Menu for Objects” on page 5-10 for more information.

- Status bar

  Displays tool tips and status information.

Drawing Objects

A state diagram comprises seven types of graphical objects: states, boxes, functions, transitions, default transitions, history junctions, and connective junctions. Stateflow provides tools for creating instances of each of these types of objects as shown in the following:



In general, you can draw a chart object by first selecting one of the tools and then click-dragging on the chart area to form the object. This rule has two exceptions:

- Transitions do not require the preselection of a drawing tool.

  The Transition tool, used to draw transitions, is available by default. You can click-drag a transition at any time without selecting an icon in the Stateflow editor's object palette.
- Junctions require only a single click to place them at the cursor's position.

For detailed information on creating Statechart objects, see the following topics:

- "Using States in Stateflow Charts" on page 5-18
- "Using Boxes in Stateflow Charts" on page 5-47
- "Creating a Graphical Function" on page 5-48
- "Using Transitions in Stateflow Charts" on page 5-28
- "Using Junctions in Stateflow Charts" on page 5-57
- "Using Notes in Stateflow Charts" on page 5-61

Displaying the Context Menu for Objects

Every object that you create in a state diagram has a shortcut menu associated with it. To display the shortcut (context) menu, do the following:

1. Move the cursor over the object.
2. Right-click the object.

   Stateflow pops up a menu of operations that apply to the object.

To display the context menu for the chart object, do the following:

1. Move the cursor to an unoccupied location in the diagram.
2. Right-click the location.

   Stateflow pops up a menu of operations that apply to the chart.

Specifying Colors and Fonts

You can specify the color and font for items in the diagram editor, as described in the following topics:

- "Changing Fonts for an Individual Text Item" on page 5-10 — Tells you how to set color and font for an individual item in the Stateflow diagram editor.
- "Using the Colors & Fonts Dialog" on page 5-11 — Shows you how to set default colors and fonts for all Stateflow diagram editor items in the Colors and Fonts dialog

Changing Fonts for an Individual Text Item

You can change the font for an individual text item as follows:

1. Right-click the individual item.
2. From the resulting submenu, select **Font Size** -> *size of font*.

You can also specify the label font size of a particular object:

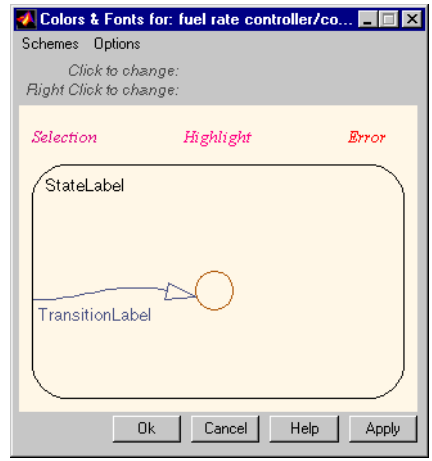
1. Left-click an individual text item in the editor.
2. From the editor's **Edit** menu, select **Set Font Size.**

3 From the resulting submenu, select the font size.

Using the Colors & Fonts Dialog

The Stateflow **Colors & Fonts** dialog allows you to specify a color scheme for a chart as a whole, or colors and label fonts for different types of objects in a chart.

To display the **Colors & Fonts** dialog, select **Style** from the Stateflow editor's **Edit** menu.



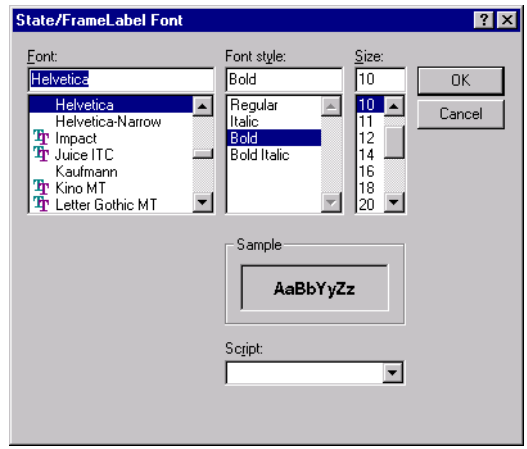
The drawing area of the dialog displays examples of the types of objects whose colors and font labels you can specify. The examples use the colors and label fonts specified by the current color scheme for the chart. To choose another color scheme, select the scheme from the dialog's **Schemes** menu. The dialog displays the selected color scheme. Click **Apply** to apply the selected scheme to the chart or **OK** to apply the scheme and dismiss the dialog.

To make the selected scheme the default scheme for all Stateflow charts, select **Make this the "Default" scheme** from the dialog's **Options** menu.

To modify the current scheme, position the cursor over the example of the type of object whose color or label font you want to change. Then left-click to change the object's color or right-click to change the object's font. If you left-click, Stateflow displays a color chooser dialog.

Use the dialog to select a new color for the selected object type.

If the selected object is a label and you right-click, Stateflow displays a font selection dialog.



Use the font selector to choose a new font for the selected label.

To save changes to the default color scheme, select **Save defaults to disk** from the **Colors & Fonts** dialog's **Options** menu.

Note Choosing **Save defaults to disk** has no effect if the modified scheme is not the default scheme.

Selecting and Deselecting Objects

Once an object is in the drawing area, you need to select it to make any changes or additions to that object. To select an object, click it. When an object is selected, it is highlighted in the color set as the selection color (blue by default; see "Specifying Colors and Fonts" on page 5-10 for more information).

To select multiple objects, click and drag a selection rubberband so that the rubberband box encompasses or touches the objects you want to select. Once all objects are within the rubberband, release the left mouse button. All objects or portions of objects within the rubberband are selected.

Simultaneously pressing the **Shift** key and clicking an object either adds that object to the selection list if it was deselected or deselects the object if it is on the selection list. This is useful for selecting objects within a state without selecting the state itself.

To select all objects in the Stateflow diagram, choose **Select All** from the **Edit** menu or the right-click shortcut menu.

Simultaneously pressing the **Shift** key and doing a rubberband selection selects objects touched by the rubberband if they are deselected and deselects objects touched by the rubberband if they are selected.

Pressing the **Esc** key deselects all selected objects. Pressing the **Esc** key again displays the parent of the current chart.

Cutting and Pasting Objects

You can cut objects from the drawing area or cut and then paste them as many times as you like. You can cut and paste objects from one Stateflow diagram to another. Stateflow retains a selection list of the most recently cut objects. The objects are pasted in the drawing area location closest to the current mouse location.

To cut an object, select the object and choose **Cut** from one of the following:

- The **Edit** menu on the main window
- The right-click shortcut menu

Pressing the **Ctrl** and **X** keys simultaneously is the keyboard equivalent to the **Cut** menu item.

To paste the most recently cut selection of objects, choose **Paste** from either of the following:

- The **Edit** menu on the main window
- The right-click shortcut menu

Pressing the **Ctrl** and **V** keys simultaneously is the keyboard equivalent to the **Paste** menu item.

Copying Objects

To copy and paste an object in the drawing area, select the objects and right-click and drag them to the desired location in the drawing area. This operation also updates the Stateflow clipboard.

Alternatively, to copy from one Stateflow diagram to another, choose the **Copy** and then **Paste** menu items from either of the following:

- The **Edit** menu on the Stateflow graphics editor window
- The right-click shortcut menu

Pressing the **Ctrl** and **C** keys simultaneously is the keyboard equivalent to the **Copy** menu item. States that contain other states (superstates) can be grouped together.

Editing Object Labels

Some Stateflow objects (for example, states and transitions) have labels. To change these labels, place the cursor anywhere in the label and click. The cursor changes to an I-beam. You can then edit the text.

The shortcut (context) menus allows you to change a label's font size:

1. Select the states whose label font size you want to change.
2. Right-click to display the shortcut menu.

3 Place the cursor over the **Font Size** menu item.

A menu of font sizes appears.

4 Select the desired font size from the menu.

Stateflow changes the font size of all labels on all selected states to the selected size.

Viewing Data and Events from the Editor

To view or modify events and data defined by any state visible in the Stateflow editor window (see "Defining Events and Data" on page 6-1), position the mouse cursor over the state, right-click to display the state's context menu, and select **Explore** from the context menu. Stateflow opens the Stateflow Explorer (if not already open) and expands its object hierarchy view (see "Explorer Main Window" on page 10-4) to show any events or data defined by the state.

To view events and data defined by a transition or junction's parent state, select **Explore** from the transition or junction's context menu.

Zooming a Diagram

You can magnify or shrink a diagram, using the following zoom controls:

- **Zoom Factor Selector**. Selects a zoom factor (see "Using the Zoom Factor Selector" on page 5-15).
- **Zoom In** button. Zooms in by the current zoom factor.

  You can also press the **R** key to increase the zoom factor.
- **Zoom Out** button. Zooms out by the current zoom factor.

  You can also press the **V** key to decrease the zoom factor.

Using the Zoom Factor Selector

The **Zoom Factor Selector** allows you to specify the zoom factor by

- Choosing a value from a menu.

  Click the selector to display the menu.

- Double-clicking the **Zoom Factor Selector** selects the zoom factor that will fit the view to all selected objects or all objects if none are selected.

  You can achieve the same effect by choosing **Fit to View** from the right-click context menu or by pressing the **F** key to apply the maximum zoom that includes all selected objects. Press the space bar to fit all objects to the view.
- Clicking the **Zoom Factor Selector** and dragging up or down.

  Dragging the mouse upward increases the zoom factor. Dragging the mouse downward decreases the zoom factor. Alternatively, right-clicking and dragging on the percentage value resizes while you are dragging.

Zooming with Shortcut Keys

The following is a summary of the shortcut keys you can use to perform some of the zooming operations described above:

Key	Description
F	Highlight (select) an object and press **F** to fit it to view.
space bar	Set to full view of diagram.
R	Increase zoom factor.
V	Decrease zoom factor.

Undoing and Redoing Editor Operations

You can undo and redo operations you perform in the Stateflow diagram editor. When you undo an operation in the Stateflow diagram editor, you reverse the last edit operation you performed. After you undo operations in the diagram editor, you can also redo them one at a time.

To undo an operation in the Stateflow diagram editor, do one of the following:

- Select the **Undo** icon in the toolbar of the Stateflow diagram editor .

  When you place your mouse cursor over the **Undo** button, the tool tip that appears indicates the nature of the operation to undo.
- From the **Edit** menu, select **Undo.**

To redo an operation in the Stateflow diagram editor, do one of the following:

- Select the **Redo** icon in the toolbar of the Stateflow diagram editor .

  When you place your mouse cursor over the **Redo** button, the tool tip that appears indicates the nature of the operation to redo.
- From the **Edit** menu, select **Redo.**

Exceptions for Undo

You can undo or redo all diagram editor operations, with the following exceptions:

- You cannot undo the operation of turning subcharting off for a state previously subcharted.

  To understand subcharting, see "Using Subcharts in Stateflow Charts" on page 5-64.
- You cannot undo the drawing of a supertransition or the splitting of an existing transition.

  Splitting of an existing transition refers to the redirection of the source or destination of a transition segment that is part of a supertransition. For a description of supertransitions, see "Drawing a Supertransition" on page 5-72.
- You cannot undo any changes made to the diagram editor through the Stateflow API.

  For a description of the Stateflow API (Application Programming Interface), see "Overview of the Stateflow API" on page 13-3.

Caution When you perform one of the preceding operations, the undo and redo buttons are disabled from undoing and redoing any prior operations.

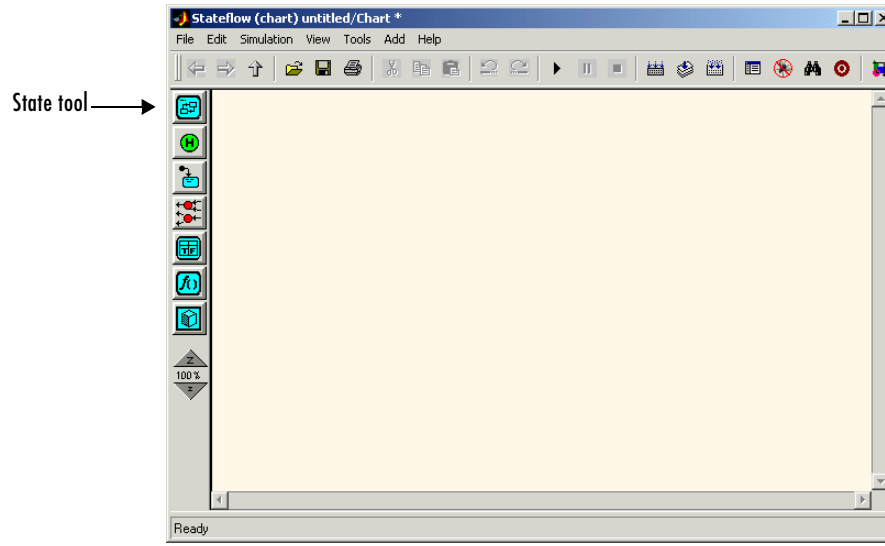
Using States in Stateflow Charts

Stateflow diagrams react to events by changing states, which are modes of a chart. This section describes how to create and specify a state in your new chart and contains the following topics:

- "Creating a State" on page 5-18 — Shows you how to draw states in the Stateflow diagram editor with the State drawing tool
- "Moving and Resizing States" on page 5-19 — Tells you how to move and resize states in the diagram editor
- "Creating Substates" on page 5-20 — Tells you how to create substates of owning superstates
- "Grouping States" on page 5-20 — Tells you how to group (and ungroup) states, boxes, and functions for convenience
- "Specifying Substate Decomposition" on page 5-21 — Tells you how to specify the substate decomposition type (parallel or exclusive) for a state or chart
- "Specifying Activation Order for Parallel States" on page 5-21 — Tells you how to specify the order of activation for parallel (AND) substates in a state or chart with parallel (AND) decomposition
- "Changing State Properties" on page 5-22 — Tells you how to change the properties for a state
- "Labeling States" on page 5-23 — Tells you how to label a state with its name and actions
- "Outputting State Activity to Simulink" on page 5-26 — Tells you how to output a state's activity as an output port signal on a Stateflow block

Creating a State

You create states by drawing them in the Stateflow diagram editor for a particular Stateflow chart (block). The following is a depiction of the Stateflow diagram editor:



To activate the State tool, click or double-click the **State** button in the Stateflow toolbar. Single-clicking the button puts the State tool in single-creation mode. In this mode, you create a state by clicking the tool in the drawing area. Stateflow creates the state at the specified location and returns to edit mode.

Double-clicking the **State** button puts the State tool in multiple-creation mode. This mode works the same way as single-creation mode except that the State tool remains active after you create a state. You can thus create as many states as you like in this mode without having to reactivate the State tool. To return to edit mode, click the **State** button, or right-click in the drawing area, or press the **Esc** key.

To delete a state, select it and choose **Cut** (**Ctrl+X**) from the **Edit** or any shortcut menu or press the **Delete** key.

Moving and Resizing States

To move a state, do the following:

1. Left-click and drag the state.
2. Release it in a new position.

To resize a state, do the following:

1 Place the cursor over a corner of the state.

  When the cursor is over a corner, it appears as a double-ended arrow (PC only; cursor appearance varies with other platforms).

2 Left-click and drag the state's corner to resize the state.

3 Release the left mouse button.

Creating Substates

A substate is a state that can be active only when another state, called its parent, is active. States that have substates are known as superstates. To create a substate, click the State tool and drag a new state into the state you want to be the superstate. Stateflow creates the substate in the specified parent state. You can nest states in this way to any depth. To change a substate's parentage, drag it from its current parent in the state diagram and drop it in its new parent.

Note A parent state must be large enough to accommodate all its substates. You might need to resize a parent state before dragging a new substate into it.

Grouping States

Grouping a state causes Stateflow to treat the state and its contents as a graphical unit. This simplifies editing a state diagram. For example, moving a grouped state moves all its substates as well.

To group a state, do one of the following:

- Double-click the state or its border
- Right-click the state. From the resulting popup menu select **Make Contents** and then **Grouped** from the resulting submenu.

Stateflow thickens the grouped state's border and grays its contents to indicate that it is grouped. To ungroup a state, double-click it or its border or select

Ungrouped from the **Make Contents** submenu units context menu. You must ungroup a superstate to select objects within the superstate.

Specifying Substate Decomposition

You specify whether a superstate contains parallel (AND) states or exclusive (OR) states by setting its decomposition. A state whose substates are all active when it is active is said to have parallel (AND) decomposition. A state in which only one substate is active when it is active is said to have exclusive (OR) decomposition. An empty state's decomposition is exclusive.

To alter a state's decomposition, select the state, right-click to display the state's shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** from the menu.

You can also specify the state decomposition of a chart. In this case, Stateflow treats the chart's top-level states as substates of the chart. Stateflow creates states with exclusive decomposition. To specify a chart's decomposition, deselect any selected objects, right-click to display the chart's shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** from the menu.

The appearance of a superstate's substates indicates the superstate's decomposition. Exclusive substates have solid borders, parallel substates, dashed borders. A parallel substate also contains a number in its upper right corner. The number indicates the activation order of the substate relative to its sibling substates.

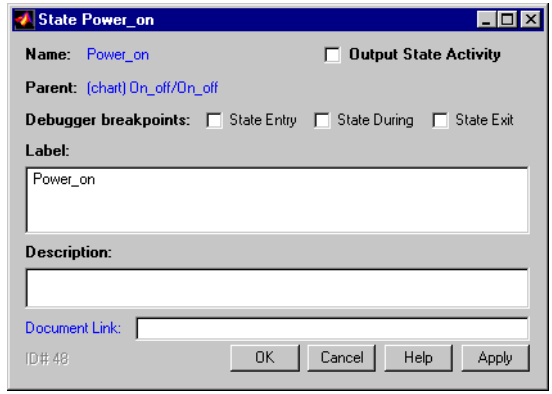
Specifying Activation Order for Parallel States

You specify the activation order of parallel states by arranging them from top to bottom and left to right in the state diagram. Stateflow activates the states in the order in which you arrange them. In particular, a top-level parallel state activates before all the states whose top edges reside at a lower level in the state diagram. A top-level parallel state also activates before any other state that resides to the right of it at the same vertical level in the diagram. The same top to bottom, left to right activation order applies to parallel substates of a state.

Note Stateflow displays the activation order of a parallel state in its upper right corner.

Changing State Properties

The **State Properties** dialog box lets you view and change a state's properties. To display the dialog for a particular state, choose **Properties** from the state's shortcut menu or click the state's entry in the Explorer contents pane. Stateflow displays the state's properties dialog box.



The dialog includes the following fields.

Field	Description
Name	Stateflow diagram name; read-only; click this hypertext link to bring the state to the foreground.
Output State Activity	Select this check box to cause Stateflow to output the activity status of this state to Simulink via a data output port on the Chart block containing the state. See "Outputting State Activity to Simulink" on page 5-26 for more information.

Field	Description
Parent	Parent of this state; a / character indicates the Stateflow diagram is the parent; read-only; click this hypertext link to bring the parent to the foreground.
Debugger breakpoints	Click the check boxes to set debugging breakpoints on state `entry`, `during`, or `exit` actions. See Chapter 12, "Debugging and Testing," for more information.
Label	The state's label. See the section titled "Labeling States" on page 5-23 for more information.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit /spec/data/speed.txt`.

Click the dialog **Apply** button to save the changes. Click the **Revert** button to cancel any changes and return to the previous settings. Click the **Close** button to save the changes and close the dialog box. Click the **Help** button to display the Stateflow online help in an HTML browser window.

Labeling States

Every state has a label. A state's label specifies its name and the actions executed when the state is entered, exited, or receives an event while it is active. How to edit the label and the contents it requires are described in the topics that follow.

- "Editing a State Label" on page 5-24
- "State Label Format" on page 5-24
- "Entering the Name" on page 5-25
- "Entering Entry Actions" on page 5-25
- "Entering During Actions" on page 5-26

- "Entering Exit Actions" on page 5-26
- "Entering On Event_Name Actions" on page 5-26

For more information on state label concepts, see the following:

- "State Label Notation" on page 3-10
- "Action Language" on page 7-1

Editing a State Label

Initially, a state's label is empty. Stateflow indicates this by displaying a ? in the state's label position (upper left corner).

Label unlabeled states as follows:

1. Select (left-click) the state.

   The state turns to its highlight color.

2. Left-click the ? to edit the label.

   An editing cursor appears. You are now free to type a label.

To apply and exit the edit, deselect the object. To reedit the label, simply left-click the label text near the character position you want to edit.

You can also edit the state's label through the properties dialog for the state. See "Changing State Properties" on page 5-22.

State Label Format

State labels have the following general format.

```
name/
entry:entry actions
during:during actions
exit:exit actions
on event_name:on event_name actions
```

The italicized entries in this format have the following meanings:

Entry	Description
name	A unique reference to the state with optional slash
entry actions	Actions executed when a particular state is entered as the result of a transition taken to that state
during actions	Actions that are executed when a state receives an event while it is active with no valid transition away from the state
exit actions	Actions executed when a state is exited as the result of a transition taken away from the state
event_name	A specified event
on event_name actions	Actions executed when a state is active and the specified event *event_name* occurs See "Defining Events" on page 6-2 for information on defining and using events.

Entering the Name

Enter the state's name in the first line of the state's label. Names are case sensitive. To avoid naming conflicts, do not assign the same name to sibling states. However, you can assign the same name to states that do not share the same parent.

Entering Entry Actions

Entry actions begin on a new line and consist of the keyword `entry` (or just `en`), followed by a colon, followed by one or more action statements on one or more lines. You must separate statements on the same line by a comma or semicolon.

Note You can also begin a state's entry action on the same line as the state's name. In this case, begin the entry action with a forward slash (/) instead of the entry keyword.

Entering During Actions

During actions have the same format as entry actions except that they begin with the keyword `during` or `dur`.

Entering Exit Actions

Exit actions have the same format as entry actions except that they begin with the keyword `exit` or `ex`.

Entering On Event_Name Actions

`On` *`event_name`* actions specify an event handler for a state. They have the same format as an entry actions except that they begin with the keyword `on`, followed by the name of the event, followed by a colon, for example:

```
on ev1: exit();
```

If you want different events to trigger different actions, enter multiple `on` *`event_name`* blocks in the state's label, each specifying the action for a particular event or set of events, for example:

```
on ev1: action1(); on ev2: action2();
```

Note Use a `during` block for a state to specify actions to take in response to any visible event that occurs while that state is active but without a valid transition to another state (see "Entering During Actions" on page 5-26).

Outputting State Activity to Simulink

Stateflow allows a chart to output the activity of its states to Simulink via a data port on the state's Chart block. To enable output of a particular state's activity, first name the state (see "Entering the Name" on page 5-25), if unnamed, then select the **Output State Activity** check box on the state's property dialog (see "Changing State Properties" on page 5-22). Stateflow

creates a data output port on the Chart block containing the state. The port has the same name as the state. Stateflow also adds a corresponding data object of type State to the Stateflow data dictionary. During simulation of a model, the port outputs 1 at each time step in which the state is active; 0, otherwise. Attaching a scope to the port allows you to monitor a state's activity visually during the simulation. See "Defining Output Data" on page 6-27 for more information.

Note If a chart has multiple states with the same name, only one of those states can output activity data. If you check the `Output State Activity` property for more than one state with the same name, Stateflow outputs data only from the first state whose `Output State Activity` property you specified.

Using Transitions in Stateflow Charts

Charts change active states using pathways called transitions. This section describes how to create, move, change, and specify properties for Stateflow transitions with the following topics:

- "Creating a Transition" on page 5-28 — Tells you how to create transitions between states and junctions.
- "Creating Straight Transitions" on page 5-29 — Gives you the methods for straightening curved transitions.
- "Labeling Transitions" on page 5-30 — Tells you how to edit the default ? label for each transition and the correct format for a transition label.
- "Moving Transitions" on page 5-31 — Tells you how to move the line, the attach points, and the label of a transition.
- "Changing Transition Arrowhead Size" on page 5-33 — Gives you the procedures for changing the arrowhead size of transitions.
- "Creating Self-Loop Transitions" on page 5-33 — Shows you how to create self-loop transitions that loop back to their source.
- "Creating Default Transitions" on page 5-34 — Tells you how to create default transitions.
- "Setting Smart Behavior in Transitions" on page 5-34 — Shows you how to give a transition "smart" behavior.
- "What Smart Transitions Do" on page 5-35 — Shows you the special behavior exhibited by "smart" transitions and why it might be useful.
- "What Nonsmart Transitions Do" on page 5-42 — Shows you the behavior of nonsmart transitions and why it might be useful.
- "Changing Transition Properties" on page 5-44 — Tells you how to inspect and change some of the properties of a transition through its properties dialog.

Creating a Transition

Use the following procedure for creating transitions between states and junctions:

1 Place the cursor on or close to the border of a source state or junction.

   The cursor changes to crosshairs.

2 Click and drag a transition to a destination state or junction.

3 Release on the border of the destination state or junction.

   Notice that the source of the transition sticks to the initial source point for the transition.

Attached transitions obey the following rules:

- Transitions do not attach to the corners of states. Corners are used exclusively for resizing.
- Transitions exit a source and enter a destination at angles perpendicular to the source or destination surface.
- Newly created transitions have smart behavior. See "Setting Smart Behavior in Transitions" on page 5-34.

To delete a transition, select it and choose **Ctrl+X** or **Cut** from the **Edit** menu, or press the **Delete** key.

See the following sections for help with creating *self-loop* and *default* transitions:

- "Creating Self-Loop Transitions" on page 5-33
- "Creating Default Transitions" on page 5-34

Creating Straight Transitions

While creating a transition, notice that the source of the transition sticks to the initial source point. This often results in a curved transition. To create a perfectly straight transition, while click-dragging from one state to another, do one of the following:

- Press the **S** key (works on all platforms).
- Right-click the mouse (works on *most* platforms).

Either of these actions straightens the transition perpendicular to the transition's source state or junction surface, if possible, and allows the transition source point to slide to maintain straightness. For states, if the

cursor is out of range of perpendicularity with the source state, the transition is unaffected.

Labeling Transitions

Transition labels contain Stateflow action language that accompanies the execution of a transition. Creating and editing transition labels is described in the following topics:

- "Editing Transition Labels" on page 5-30
- "Transition Label Format" on page 5-30

For more information on transition concepts, see "Transition Label Notation" on page 3-15.

For more information on transition label contents, see "Action Language" on page 7-1.

Editing Transition Labels

Label unlabeled transitions as follows:

1. Select (left-click) the transition.

   The transition turns to its highlight color and a question mark (?) appears on the transition. The ? character is the default empty label for transitions.

2. Left-click the ? to edit the label.

   An editing cursor appears. You are now free to type a label.

To apply and exit the edit, deselect the object. To reedit the label, simply left-click the label text near the character position you want to edit.

Transition Label Format

Transition labels have the following general format:

```
event [condition]{condition_action}/transition_action
```

Specify, as appropriate, relevant names for `event`, `condition`, `condition_action`, and `transition_action`.

Label Field	Description
`event`	Event that causes the transition to be evaluated.
`condition`	Defines what, if anything, has to be true for the condition action and transition to take place.
`condition_action`	If the condition is true, the action specified executes and completes.
`transition_action`	This action executes after the source state for the transition is exited but before the destination state is entered.

Transitions do not have to have labels. You can specify some, all, or none of the parts of the label. Valid transition labels are defined by the following:

- Can have any alphanumeric and special character combination, with the exception of embedded spaces
- Cannot begin with a numeric character
- Can have any length
- Can have carriage returns in most cases
- Must have an ellipsis (...) to continue on the next line

Moving Transitions

You can move transition lines with a combination of several individual movements. These movements are described in the following topics:

- “Bowing the Transition Line” on page 5-32
- “Moving Transition Attach Points” on page 5-32
- “Moving Transition Labels” on page 5-32

In addition, transitions move along with the movements of states and junctions. See “Setting Smart Behavior in Transitions” on page 5-34 for a description of *smart* and *nonsmart* transition behavior.

Bowing the Transition Line

You can move or "bow" transition lines with the following procedure:

1. Place the cursor on the transition at any point along the transition except the arrow or attach points.
2. Click and drag the mouse to move the transition point to another location.

   Only the transition line moves. The arrow and attachment points do not move.
3. Release the mouse button to specify the transition point location.

The result is a bowed transition line. Repeat the preceding steps to move the transition back into its original shape or into another shape.

Moving Transition Attach Points

You can move the source or end points of a transition to place them in exact locations as follows:

1. Place the cursor over an attach point until it changes to a small circle.
2. Click and drag the mouse to move the attach point to another location.
3. Release the mouse button to specify the new attach point.

The appearance of the transition changes from a solid to a dashed line when you detach and release a destination attach point. Once you attach the transition to a destination, the dashed line changes to a solid line.

The appearance of the transition changes to a default transition when you detach and release a source attach point. Once you attach the transition to a source, the appearance returns to normal.

Moving Transition Labels

You can move transition labels to make the Stateflow diagram more readable. To move a transition label, do the following:

1. Click and drag the label to a new location.

2 Release the left mouse button.

If you mistakenly click and then immediately release the left mouse button on the label, you will be in edit mode for the label. Press the **Esc** key to deselect the label and try again. You can also click the mouse on an empty location in the diagram editor to deselect the label.

Changing Transition Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of an object causes all incoming arrowheads to that object to be adjusted to the same size. The arrowhead size of any selected transitions, and any other transitions ending at the same object, is adjusted.

To adjust arrowhead size from the **Transition** shortcut menu:

1 Select the transitions whose arrowhead size you want to change.

2 Place the cursor over a selected transition and right-click to display the shortcut menu.

A menu of arrowhead sizes appears.

3 Select an arrowhead size from the menu.

To adjust arrowhead size from the **Junction** shortcut menu:

1 Select the junctions whose incoming arrowhead size you want to change.

2 Place the cursor over a selected junction and right-click.

3 In the resulting submenu, place the cursor over **Arrowhead Size**.

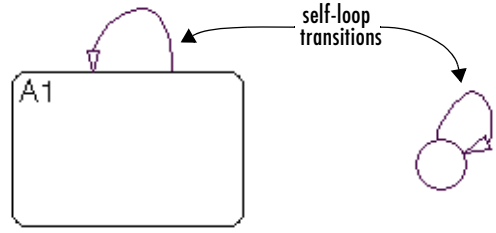
A menu of arrowhead sizes appears.

4 Select a size from the menu.

Creating Self-Loop Transitions

A self-loop transition is a transition whose source and destination are the same state or junction.

The following is an example of a self-loop transition:



To create a self-loop transition, do the following:

1 Create the transition as usual by click-dragging it out from the source state or junction.

2 Continue dragging the transition tip back to a location on the source state or junction.

For the semantics of self-loops, see "Self-Loop Transitions" on page 3-22.

Creating Default Transitions

A default transition is a transition with a destination (a state or a junction), but no apparent source object. See "Default Transitions" on page 2-14 for an explanation of default transitions.

Click the **Default Transition** button in the toolbar and click a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition.

The size of the endpoint of the default transition is proportional to the arrowhead size. See "Changing Transition Arrowhead Size" on page 5-33.

Default transitions can be labeled just like other transitions. See "Labeling Default Transitions" on page 3-27 for an example.

Setting Smart Behavior in Transitions

Transitions with smart behavior, that is, smart transitions, allow their ends to slide around the surfaces of Stateflow objects. This allows transitions to maintain their shapes and uniqueness while you rearrange chart objects.

Without this ability, rearranging a Stateflow chart full of states and junctions attached by transitions can soon become a very arduous task. Now, objects move but stay attached in cleaner and more efficient ways. The result is a chart that is cleaner and easier to rearrange.

Transitions are automatically created with smart behavior, on the assumption that this behavior is desirable in most circumstances. You can disable or enable smart behavior in existing transitions with the following procedure:

1 Right-click a transition.

   On the resulting menu, observe the selection titled **Smart**. If a check mark appears in front of **Smart**, the transition has smart behavior.

2 If **Smart** is not checked, select it to enable smart behavior.

   To disable smart transition behavior, select **Smart** if it is already checked.

See the following sections for a comparison of behavior between smart and nonsmart transitions:

- "What Smart Transitions Do" on page 5-35
- "What Nonsmart Transitions Do" on page 5-42

Note Transitions with smart behavior differ graphically only. Apart from graphical behavior, there is no difference in meaning between a transition with and without smart behavior.

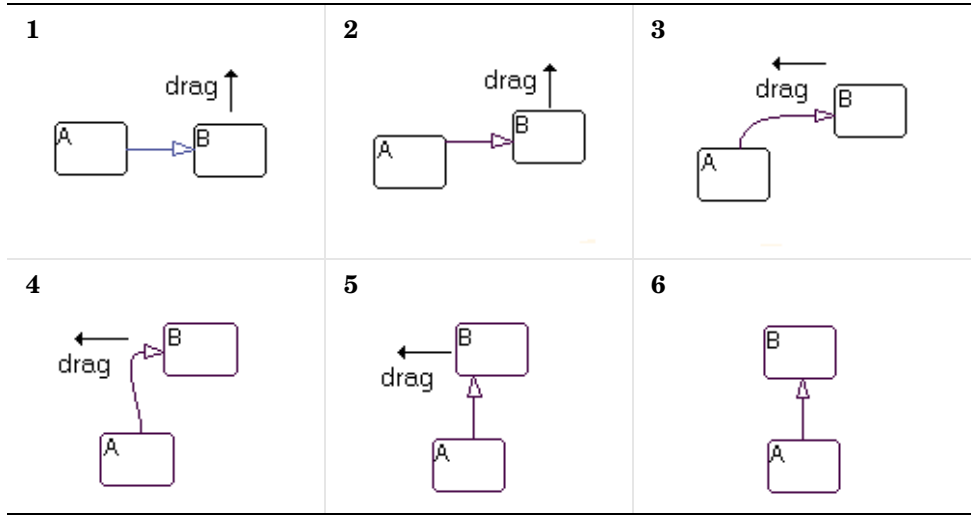
What Smart Transitions Do

The following topics discuss some of the behaviors of smart transitions:

- "Smart Transitions Slide Around Surfaces" on page 5-36
- "Smart Transitions Slide and Maintain Shape" on page 5-37
- "Smart Transitions Connect States to Junctions at 90 Degree Angles" on page 5-38
- "Smart Transitions Snap to an Invisible Grid" on page 5-40
- "Smart Transitions Bow Symmetrically" on page 5-41

Smart Transitions Slide Around Surfaces

In the following example, state B is attached to state A by a smart transition. The example shows state B being dragged counterclockwise around the upper right corner of state A. When this occurs, state B turns to its selection color and the transition turns to a very light shade of gray, a sure sign of smart behavior. Dragging direction is shown by the arrows.



Note the following step-by-step behavior for the preceding example:

1 The first capture shows states A and B at the beginning of movement.

2 As B moves upward, the transition's back end slides upward on A, maintaining the transition straight.

3 As B moves around A's corner, the back end of the transition suddenly hops around A's upper right-hand corner. The transition is now curved from A's top surface to B's left side, maintaining perpendicularity with each attached state side.

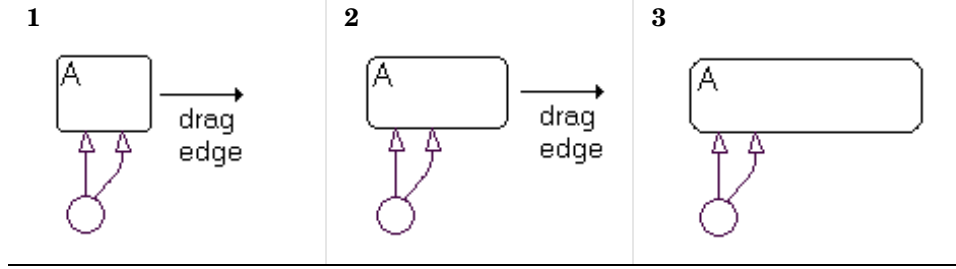
Note A hop around a state's corner is a necessity because transitions are restricted from attaching at corners of states.

4. As B moves on top of A, the transition stays curved but its front end slides down to B's lower left-hand corner.
5. As B continues to move to the left over A, the transition's front end hops around B's lower left-hand corner.
6. Finally, as B moves directly over A, the transition's front end slides onto B's bottom edge.

As B continues to circle A, steps 1 through 6 repeat for each of A's remaining sides.

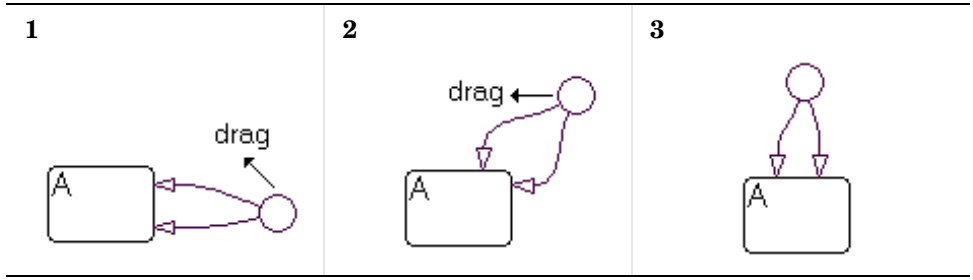
Smart Transitions Slide and Maintain Shape

While transitions with smart behavior allow their ends to slide around the surfaces of their connected objects, they also attempt to maintain their original shape during moving. In the following example, a pair of transitions with smart behavior slide during a resizing to maintain their original shape.



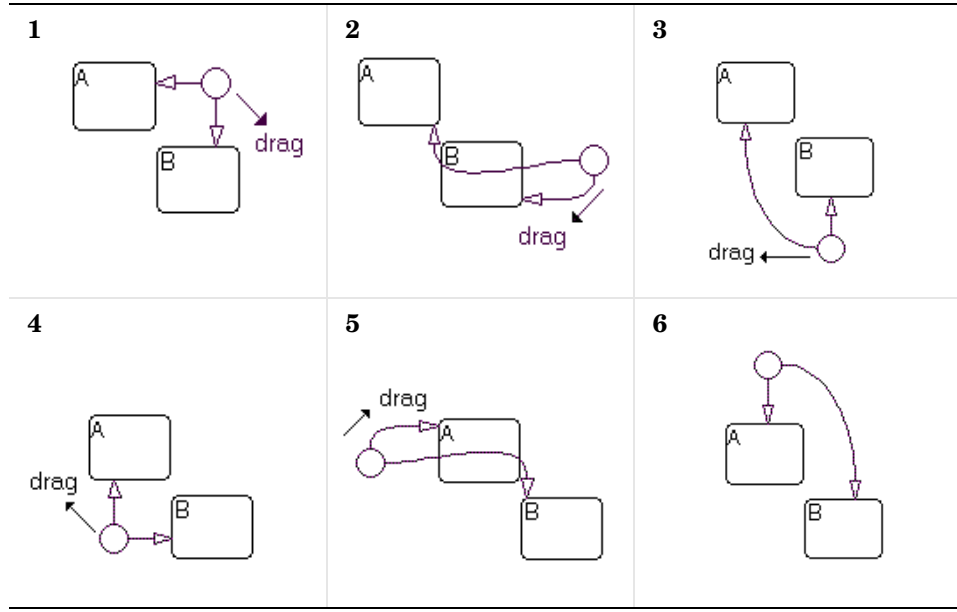
In the following example, the ends of a pair of transitions with smart behavior emanate from a junction and terminate in a state. As the junction is dragged

around the state, the ends slide around the state and maintain the same relative spacing between each other. Direction is indicated by the arrows.



Smart Transitions Connect States to Junctions at 90 Degree Angles

Straight-line connections to states must be in one of four directions: left, right, up, or down. To maintain their straightness, smart transitions from junctions always seek to connect to a state through equivalent locations on the junction (left, right, top, bottom). In the following example, a junction is connected to two states, A and B. Watch the behavior of two straight smart transitions as the junction is moved to different locations.



1 The junction starts with two straight smart transition connections to states A and B.

2 Stateflow chooses to connect the junction to state A through the junction's left side. Since the junction is below A, only a curved connection is possible.

State B could be connected by a straight line through the junction's left side, but this is already occupied by the connection to A. Therefore, B is connected through the junction's bottom, and must be curved.

3 Stateflow connects the junction to B by a straight transition through the junction's top connection. No straight-line connection to A is possible, therefore the junction is connected to state A with a curved transition through its left side.

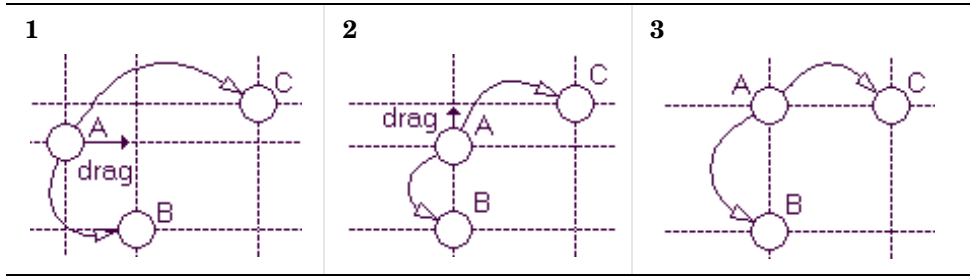
4 At this location (under A, to the left of B), straight-line transitions to A and B are possible from the junction's top and right connection points, respectively.

5 At the location left of state A, Stateflow chooses to connect to state B through its right connection point. Since the junction is above B, only a curved connection is possible.

6 Above A, a straight-line transition to state A is possible through the junction's bottom connector. A straight-line connection to state B is not possible, so the junction is connected to state B through a curved transition from its right connection.

Smart Transitions Snap to an Invisible Grid

Junctions that are connected to other junctions with smart transitions will snap to an invisible grid consisting of horizontal and vertical lines that pass through the center of each junction. The following example depicts this behavior.



Here, the invisible grid is depicted for each of the three junctions by dashed vertical and horizontal lines. Each junction is connected to each other through nonlinear smart transitions:

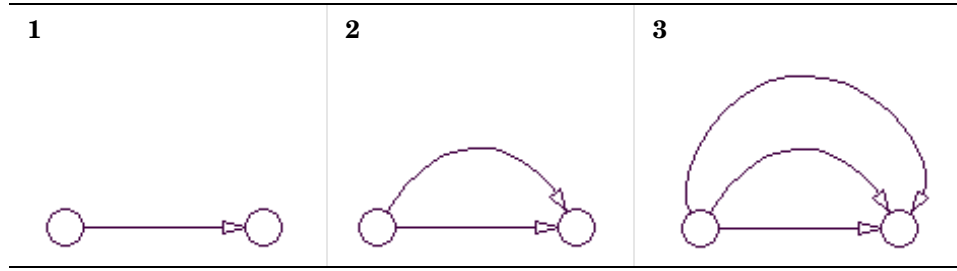
1 In the first scene, the snap grid for each junction does not overlap. The arrow indicates that junction A is being moved toward the vertical snap line for junction B.

2 When A is within a very small distance of B's snap line, A snaps into position directly above B and centered in its vertical snap line. The arrow indicates that A is now being moved toward the horizontal snap line for junction C.

3 When A is within a very small distance of C's horizontal snap line, A snaps into position directly to the side of C and centered in its horizontal snap line.

Smart Transitions Bow Symmetrically

Transitions with smart behavior bow symmetrically between junctions. In the following examples, transitions with smart behavior are drawn between two junctions:

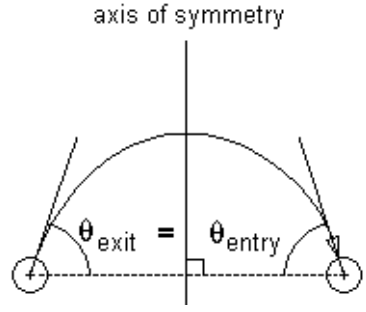


1 In the first case, a transition originates at the junction on the left and terminates on the left side of the right junction. This results in a straight line.

2 In the second case, a transition originates at the junction on the left and terminates on the top of the right junction. This results in a transition line bowed up.

3 In the third case, a transition originates at the junction on the left and terminates on the right side of the right junction. This results in a transition line bowed up even more.

Bowed smart transitions maintain symmetry by maintaining equality between transition entry and exit angles as shown below.



You can bow a smart transition between two junctions to any degree by placing the mouse cursor on any point in the transition (except the attachment points) and click-dragging in a direction perpendicular to a straight line connecting the two junctions. You can move the mouse in any direction to bow the transition but Stateflow only uses the component perpendicular to the line connecting the two junctions.

Disabling smart behavior for a transition allows you to distort the transition asymmetrically (see section "Nonsmart Transitions Distort Asymmetrically" on page 5-44). However, if you enable smart behavior again, the transition automatically returns to its prior symmetric bowed shape.

What Nonsmart Transitions Do

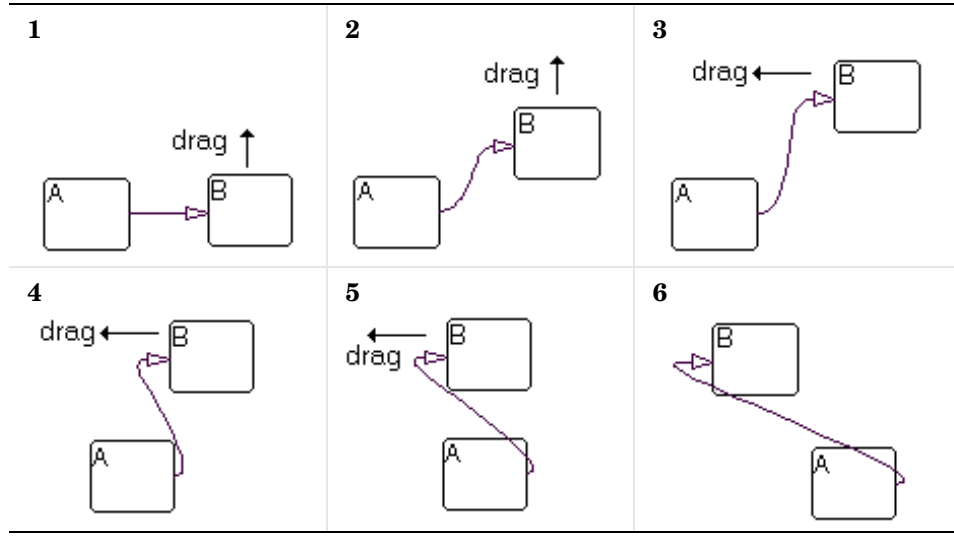
The following topics describe some of the behavior exhibited by transitions without smart behavior.

- "Nonsmart Transitions Anchor Connection Points" on page 5-43
- "Nonsmart Transitions Distort Asymmetrically" on page 5-44

You can disable and enable smart behavior in transitions. See the section "Setting Smart Behavior in Transitions" on page 5-34.

Nonsmart Transitions Anchor Connection Points

Contrast the example in the section "Smart Transitions Slide Around Surfaces" on page 5-36 with the example shown below.

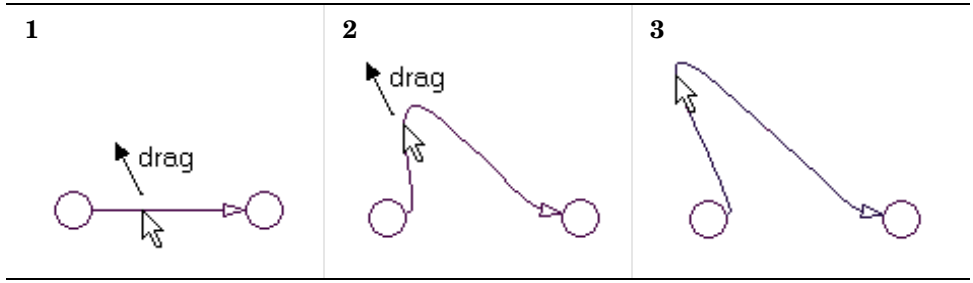


A nonsmart transition connects state A to state B. The mouse cursor is then placed over state B and left-click-dragged to new locations counterclockwise around A. When this occurs, state B turns to its highlight color but the transition remains unchanged, a sure sign of a nonsmart transition.

As B is moved around A, the transition changes into a distorted curve that seeks to maintain the original attachment points. These remain unchanged in position, although the angle of attachment is always perpendicular to the side of the state.

Nonsmart Transitions Distort Asymmetrically

Simply by click-dragging on different locations along a transition without smart behavior, you can reshape it into an asymmetric curve suited to your individual preferences. This is illustrated in the following example:

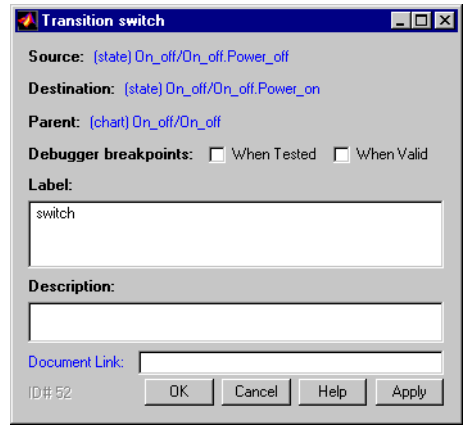


For this example, use the following procedure:

1. Drag a horizontal transition between two junctions.
2. Right-click the transition and select **Smart** from the resulting shortcut menu to disable smart behavior.
3. Place the mouse cursor on any point on the transition.
4. Click-drag the mouse cursor up and to the left.

Changing Transition Properties

Select a transition and right-click its border to display the **Transition** shortcut menu. Choose **Properties** to display the **Transition properties** dialog box.



This table lists and describes the transition object fields.

Field	Description
Source	Source of the transition; read-only; click the hypertext link to bring the transition source to the foreground.
Destination	Destination of the transition; read-only; click the hypertext link to bring the transition destination to the foreground.
Parent	Parent of this state; read-only; click the hypertext link to bring the parent to the foreground.
Debugger breakpoints	Select the check boxes to set debugging breakpoints either when the transition is tested for validity or when it is valid.
Label	The transition's label. See "Transition Label Notation" on page 3-15 for more information on valid label formats.

Field	Description
Description	Textual description/comment.
Document Link	Enter a Web URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit/spec/data/speed.txt`.

Click **Apply** to save the changes. Click **OK** to save the changes and close the dialog box. Click **Cancel** to close the dialog without applying any changes made since the last time you clicked **Apply**. Click **Help** to display the Stateflow online help in an HTML browser window.

Using Boxes in Stateflow Charts

Boxes are a convenience for organizing items in your charts. To create a box, do the following:

1. Create a state.
2. Right-click the state.
3. From the resulting submenu, select **Type -> Box**.

   Stateflow converts the state to a box, redrawing its border with sharp corners to indicate its changed status.

Once you create a box you can move or draw objects inside it. You could also draw the box first as a state around the objects you want inside it and then convert it to a box.

Like states, you can group (and ungroup) a box and its contents into a single graphical element. You can also subchart a box to hide its elements. See "Grouping States" on page 5-20 and "Using Subcharts in Stateflow Charts" on page 5-64 for more information.

For the most part, boxes do not contribute to the semantics of a Stateflow diagram. They do, however, affect the activation order of a diagram's parallel states. A box wakes up before any parallel states or boxes that are lower or to the right of it in a Stateflow diagram. Within a box, parallel states still wake up in top-down, left to right order.

Using Graphical Functions in Stateflow Charts

A *graphical function* is a function defined graphically by a flow graph that provides convenience and power to Stateflow action language. See "Graphical Functions" on page 3-41 for a more complete description.

This section describes how Stateflow graphical functions are created, called, and made available to other Stateflow charts in Simulink through the following topics:

- "Creating a Graphical Function" on page 5-48 — Shows you how to create a graphical function and describes the significance of its location.
- "Calling Graphical Functions" on page 5-53 — Shows you where and how graphical functions are called.
- "Exporting Graphical Functions" on page 5-53 — Shows you how to share the graphical function of one Stateflow chart with all the charts in a model.
- "Specifying Graphical Function Properties" on page 5-54 — Shows you how to access and set properties for graphical functions.

Creating a Graphical Function

To create a graphical function, do the following:

1 Select Graphical Function tool from the Stateflow drawing toolbar.

2 Move the cursor to the location for the new graphical function.

A function can reside anywhere in a diagram, either at the top level or within any state or subchart. The location of a function determines its scope, that is, the set of states and transitions that can invoke the function. In particular, the scope of a function is the scope of its parent state or chart, with the following exceptions.

- The chart containing the function exports its graphical functions.

  In this case, the scope of the function is the entire Stateflow machine, which encompasses all the charts in the model. See "Exporting Graphical Functions" on page 5-53 for more information.

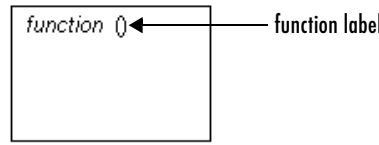
- A child of the function's parent defines a function of the same name.

  In this case, the function defined in the parent is not visible anywhere in the child or its children. In other words, a function defined in a state or

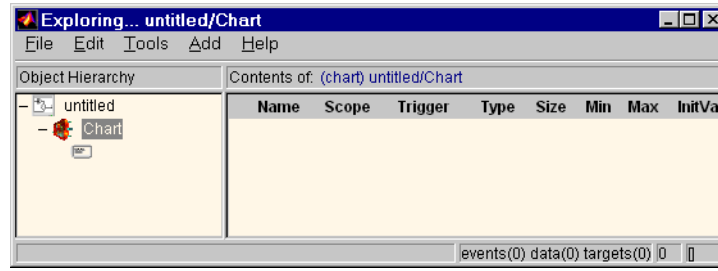
subchart overrides any functions of the same name defined in the ancestors of that state or subchart.

3 Click to place the graphical function.

The new function appears as an unnamed object in the Stateflow diagram editor.



The new function also appears in the Stateflow Explorer.

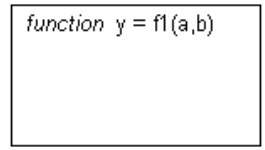


4 Edit the function signature in the function label just as you would any other label.

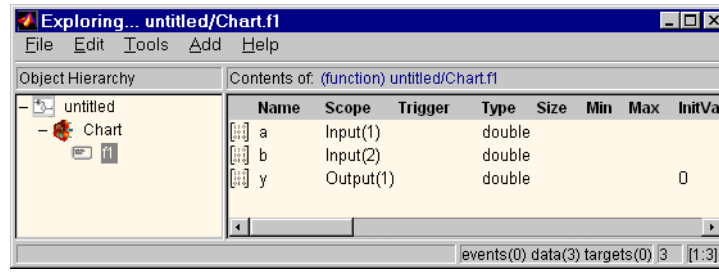
The function signature specifies a name for the function and formal names for its arguments and return value. A signature has the syntax

```
y = f(a1,a2,...an)
```

where `f` is the function's name, a_1, a_2, a_n are formal names for its arguments, and `y` is the formal name for its return value. The following example shows a signature for a graphical function named `f1` that takes two arguments and returns a value.



The return values and arguments that you declare in the signature appear in the Explorer as data items parented by the function object.



The **Scope** field in the Explorer indicates the role of the corresponding argument or return value. Arguments have scope `Input`. Return values have scope `Output`. The number that appears in parentheses for the scope of each argument is the order in which the argument appears in the function's signature. When a Stateflow action invokes a function, it passes arguments to the function in the same order.

In the context of graphical function signatures, the term *scope* refers to the role (argument or return value) of the data items specified by the function's signature. The term scope can also refer to a data item's visibility. In this sense, arguments and return values have local scope. They are visible only in the flow diagram that implements the function.

Note You can use the Stateflow editor to change the signature of a graphical function at any time. When you are done editing the signature, Stateflow updates the data dictionary and the Explorer to reflect the changes.

---

5 Specify the data properties (data type, initial value, and so on) of the function's arguments and return values (if it has any).

See "Setting Data Properties" on page 6-16 for information on setting data properties. The following restrictions apply to argument and return value properties.

- A function cannot return more than one value.
- Arguments and return values cannot be arrays.
- Arguments cannot have initial values.
- Arguments must have scope `Input`.
- Return values must have scope `Output`.

6 Create any additional data items that the function might need to process when it is invoked.

See "Adding Data to the Data Dictionary" on page 6-14 for information on how to create data items. A function can access only items that it owns. Thus, any items that you create for use by the function must be created as children of the function. The items that you create can have any of the following scopes:

- `Local`

  A local data item persists from invocation to invocation. For example, if the item is equal to 1 when the function returns from one invocation, the item will equal 1 the next time the function is invoked.

- `Temporary`

  Stateflow creates and initializes a copy of a temporary item for each invocation of the function.

- `Constant`

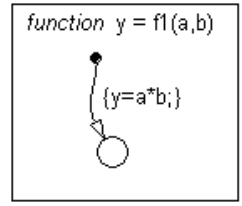
  Constant data retains its initial value through all invocations of the function.

Note You can also assign `Input` and `Output` scope to data items that you create (that is, to items that do not correspond to the function's formal arguments and return value). However, input and output items that do not correspond to your function's formal arguments and return values cause parse errors. In other words, you cannot create arguments or return values by creating data items.

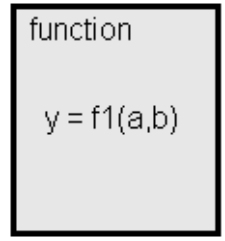
All data items (other than arguments and return values) parented by a graphical function can be initialized from the workspace. However, only local items can be saved to the workspace.

7 Create a flow diagram within the function that performs the action to be performed when the function is invoked.

At a minimum, the flow diagram must include a default transition terminated by a junction. The following example shows a minimal flow diagram for a graphical function that computes the product of its arguments.

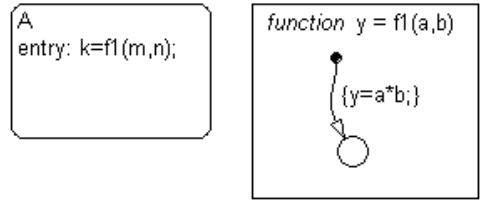


8 If you prefer, hide the function's contents by selecting **Subcharted** from the **Make Contents** menu of the function's shortcut menu.



Calling Graphical Functions

Any state or transition action that is in the scope of a graphical function can invoke that function. The invocation syntax is the same as that of the function signature, with actual arguments replacing the formal parameters specified in the signature. If the data types of the actual and formal argument differ, Stateflow casts the actual argument to the type of the formal parameter. The following example shows a state entry action that invokes a function that returns the product of its arguments.

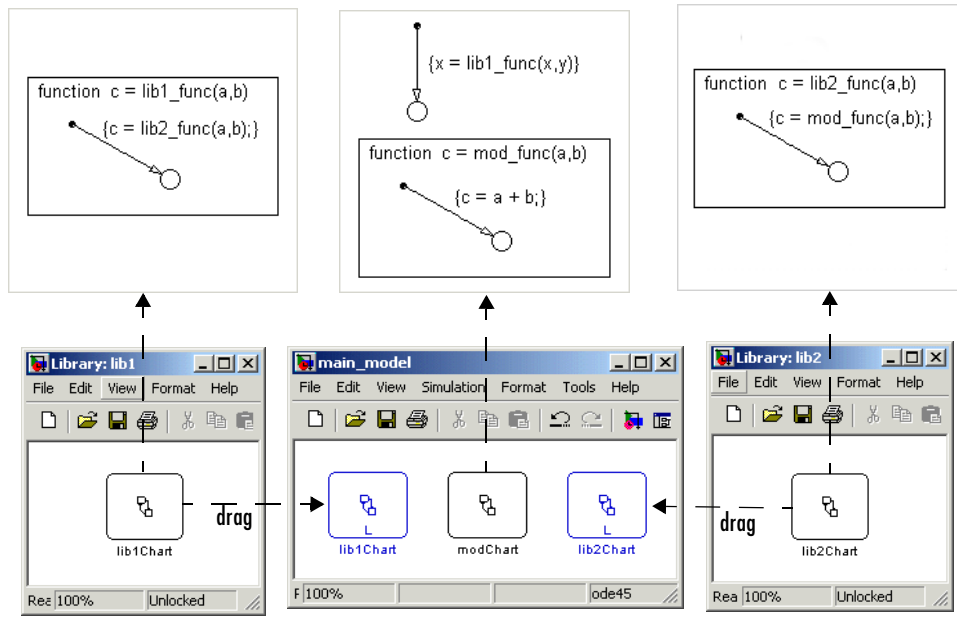


Exporting Graphical Functions

You can export a chart's root-level graphical functions to the chart's model. Exporting a chart's functions extends their scope to include all other charts in the same model.

You can also export graphical functions in library charts to a model as long as the library charts are present in the model. To export a chart's root-level functions, select **Export Chart Level Functions** on the chart's **Chart Properties** dialog box (see "Specifying Chart Properties" on page 5-78).

In the following example, the model `main_model` has two library stateflow charts, `lib1Chart` and `lib2Chart`:



Both `lib1Chart` and `lib2Chart` were dragged into the model `main_model` from the library models `lib1` and `lib2` in which they were created. In the properties dialog for all three charts, the **Export Chart Level Functions** option is selected. Each chart now defines a graphical function that can be called by any other chart placed in `main_model`.

The sequence of action in simulation of `main_model` is as follows:

- The chart `modChart` calls the graphical function `lib1_func`, with the two arguments, `x` and `y`.
- `lib1_func` calls the graphical function `lib2_func`, passing the same two arguments.
- `lib2_func` calls the graphical function `mod_func`, which adds `x` and `y`.
- The result of the addition is assigned to `x`.

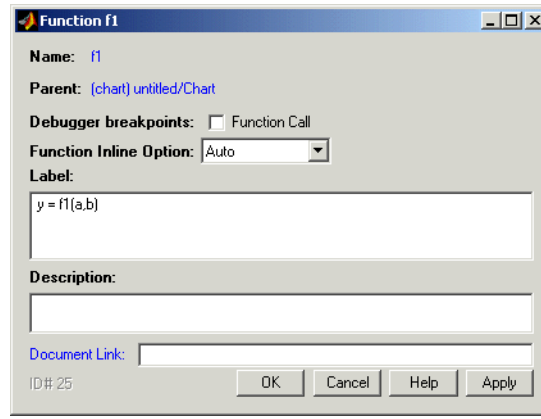
Specifying Graphical Function Properties

You can set properties for a graphical function through its shortcut menu as follows:

1 Right-click the graphical function box.

2 Select **Properties** from the resulting submenu.

The **Function Properties** dialog appears as shown:



The fields in the **Function Properties** dialog are as follows:

Field	Description
Name	Function name; read-only; click this hypertext link to bring the function to the foreground.
Parent	Parent of this function; a / character indicates the Stateflow diagram is the parent; read-only; click this hypertext link to bring the parent to the foreground.
Debugger breakpoints	Select the check box to set a breakpoint where the function is called. See “Debugging and Testing” on page 12-1 for more information.

Field	Description
Function Inline Option	This option controls the inlining of this graphical function in generated code through the following selections: • **Auto** Stateflow decides whether or not to inline the function based on an internal calculation. • **Inline** Stateflow inlines the function as long as it is not exported to other charts and is not part of a recursion. A recursion exists if the function calls itself either directly or indirectly through another called function. • **Function** The function is not inlined.
Label	The function's label. Specifies the function's signature. See "Creating a Graphical Function" on page 5-48 for more information.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit/spec/data/speed.txt`.

Using Junctions in Stateflow Charts

Junctions provide a decision point between alternate transition paths. History junctions record the activity of states inside states. The following sections describe how to create, move, and specify properties for Stateflow junctions:

- "Creating a Junction" on page 5-57 — Shows you how to create a junction in the Stateflow diagram editor
- "Changing Size" on page 5-58 — Tells you how to change the diameter of a junction in the Stateflow diagram editor
- "Moving a Junction" on page 5-58 — Shows you how to move a junction from one point to another in the Stateflow diagram editor
- "Editing Junction Properties" on page 5-58 — Tells you how to access and change the properties of a junction

Creating a Junction

To create a junction, click a **Junction** button in the toolbar and click the desired location for the junction in the drawing area. To create multiple junctions, double-click the **Junction** button in the toolbar. The button is now in multiple object mode. Click anywhere in the drawing area to place a junction in the drawing area. Additional clicks in the drawing area create additional junctions. Click the **Junction** button or press the **Esc** key to cancel the operation.

You can choose from these types of junctions.

Name	Drawing Icon	Description
Connective junction		One use of a connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.
History junction		Use a history junction in a chart or superstate to indicate that its last active substate becomes active when the chart or superstate becomes active.

Changing Size

To change the size of junctions, do the following:

1. Select the junctions whose size you want to change.
2. Place the cursor over one of the junctions and right-click.
3. In the resulting submenu, place the cursor over **Junction Size**.

   A menu of junction sizes appears.
4. Select a size from the menu of junction sizes.

Moving a Junction

To move a junction, click and drag it to a new position.

Editing Junction Properties

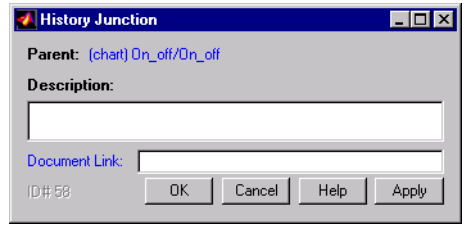
To edit the properties for a junction, do the following:

1. Right-click a junction.

2 In the resulting submenu select **Properties**.

For a connective junction, the **Connective Junction Properties** dialog box is displayed as shown.

For a history junction, the **History Junction Properties** dialog box is displayed as shown.



3 Edit the fields in the properties dialog, which are described in the following table:

Field	Description
Parent	Parent of this state; read-only; click the hypertext link to bring the parent to the foreground.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit/spec/data/speed.txt`.

4 When finished editing, select one of the following:

- Select the **Apply** button to save the changes.
- Select the **Cancel** button to cancel any changes you've made.
- Select **OK** to save the changes and close the dialog box.
- Select the **Help** button to display the Stateflow online help in an HTML browser window.

Using Notes in Stateflow Charts

Learn how to create, edit, and delete descriptive notes for your Stateflow chart with the following topics:

- "Creating Notes" on page 5-61 — Tells you how to create a chart note at any location in the Stateflow diagram editor.
- "Editing Existing Notes" on page 5-62 — Tells you how to edit an existing chart note.
- "Changing Note Font and Color" on page 5-62 — Shows you how to change the font and color of a chart note and also format it with TeX format.
- "Moving Notes" on page 5-63 — Tells you how to move a chart note from one location to another in the Stateflow diagram editor.
- "Deleting Notes" on page 5-63 — Shows you how to delete an existing chart note.

Note Chart notes are descriptive only. They do not contribute in any way to the behavior or code generation of a chart.

Creating Notes

You can enter comments/notes in any location on the chart with the following procedure:

1. Place the cursor at the desired location in the Stateflow chart.
2. Right-click the mouse.
3. From the resulting menu, select **Add Note**.

   A blinking cursor appears at the location you selected. Default text is italic, 9 point.
4. Begin typing your comments.

   As you type, the text moves left to right.
5. Press the **Return** key to start a new line.

6 When finished typing, click outside the typed note text.

Editing Existing Notes

To edit existing note text:

1 Left-click the mouse on the comment location you want to edit.

2 Once the blinking cursor appears, begin typing or use the arrow keys to move to a new text location.

Changing Note Font and Color

To change font and color for your Stateflow chart notes, follow the procedures described in the section "Specifying Colors and Fonts" on page 5-10.

You can also change your note text to bold or italic text by the doing the following:

1 Right-click the note text.

2 From the resulting shortcut menu, select **Text Format**.

3 From the resulting submenu, select **Bold** or **Italic** (default).

TeX Instructions

In the preceding procedure, note a third selection of the **Text Format** submenu called **TeX Instructions**. This selection sets the text Interpreter property to `Tex`, which allows you to use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols.

The **TeX Instructions** selection is used in the following example:

1 Right-click the text of an example note.

2 In the resulting shortcut menu, select **Text Format**.

3 In the submenu that results, make sure that **TeX Instructions** has a check mark positioned in front of it. Otherwise, select it.

4 Click the note text to place the cursor in it.

5 Replace the existing note text with the following expression.

```
\it{\omega_N = e^{(-2\pii)/N}}
```

6 Click outside the note.

The note now has the following appearance:

$\omega_N = e^{(-2\pi i)/N}$

Moving Notes

To move your notes:

1 Place the cursor over the text of the note.

2 Click and drag the note to a new location.

3 Release the left mouse button.

Deleting Notes

To delete your notes, do the following:

1 Place the mouse cursor over the text of the note.

2 Click and hold the left mouse button on the note.

A dim rectangle appears surrounding the note.

3 Select the **Delete** key.

Alternatively, you can also do the following

1 Place the mouse cursor over the text of the note.

2 Right-click the note.

3 From the resulting shortcut menu, select **Cut**.

Using Subcharts in Stateflow Charts

Subcharts are charts within charts and provide a convenience for compacting your diagrams. This section shows you how to create and work with subcharts in the following topics:

- "What Is a Subchart?" on page 5-64 — Describes a subchart and its hierarchical position in a chart.
- "Creating a Subchart" on page 5-66 — Shows you how to create a subchart by converting a state, box, or graphical function into a subchart.
- "Manipulating Subcharts as Objects" on page 5-68 — Tells you how to move, copy, cut, paste, relabel, and resize subcharts.
- "Opening a Subchart" on page 5-68 — Tells you how to open a subchart to view and change its contents.
- "Editing a Subchart" on page 5-69 — Describes the way in which you edit the contents of subcharts.
- "Navigating Subcharts" on page 5-69 — Describes a set of buttons for navigating a chart's subchart hierarchy.

What Is a Subchart?

Stateflow allows you to create charts within charts. A chart that is embedded in another chart is called a *subchart*. The subchart can contain anything a top-level chart can, including other subcharts. In fact, you can nest subcharts to any level.

A subchart is a superstate of the states and charts that it contains. It appears as a block with its name in the block center. However, you can define actions and default transitions for subcharts just as you can for superstates. You can also create transitions to and from subcharts just as you can create transitions to and from superstates. Further, you can create transitions between states residing outside a subchart and any state within a subchart. The term *supertransition* refers to a transition that crosses subchart boundaries in this way. See "Using Supertransitions in Stateflow Charts" on page 5-71 for more information.

Subcharts enable you to reduce a complex chart to a set of simpler, hierarchically organized diagrams. This makes the chart easier to understand and maintain. Nor do you have to worry about changing the semantics of the

chart in any way. Stateflow ignores subchart boundaries when simulating and generating code from Stateflow models.

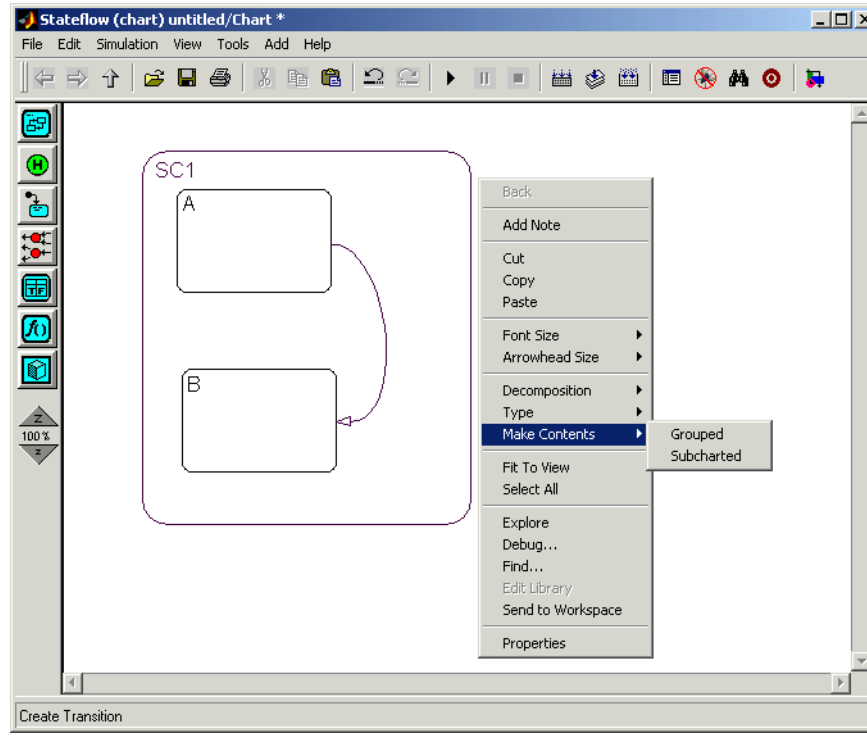
Subcharts define a containment hierarchy within a top-level chart. A subchart or top-level chart is said to be the *parent* of the charts it contains at the first level and an *ancestor* of all the subcharts contained by its children and their descendants at lower levels.

Creating a Subchart

You create a subchart by converting an existing state, box, or graphical function into the subchart. The object to be converted can be one that you have created expressly for the purpose of making a subchart or it can be an existing object whose contents you want to turn into a subchart.

To convert a new or existing state, box, or graphical function to a subchart:

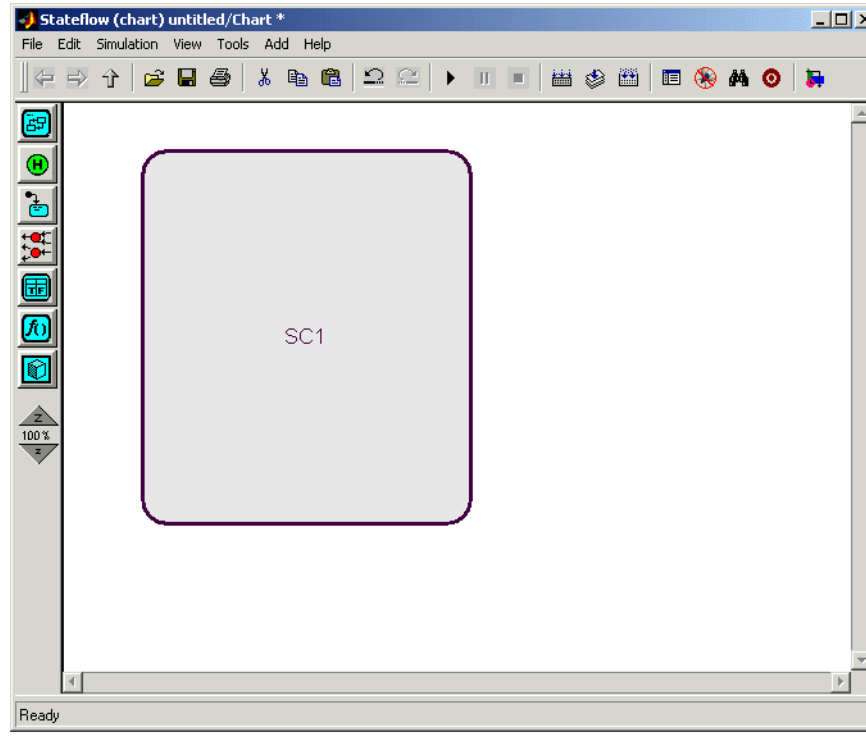
1 Select the object and right-click a state (`SC1` in the example below) to display the Stateflow shortcut menu for that state.



2 Select **Make Contents** from the resulting menu.

3 Select **Subcharted** from the resulting submenu.

Stateflow converts the state (or a graphical function or box) to a subchart.



Note When you convert a box to a subchart, the subchart retains the attributes of a box. In particular, the resulting subchart's position in the chart determines its activation order (see "Using Boxes in Stateflow Charts" on page 5-47 for more information).

To convert the subchart back to its original form, right-click the subchart. In the pop-up menu that results, select **Make Contents**. In the resulting submenu select the **Subcharted** item.

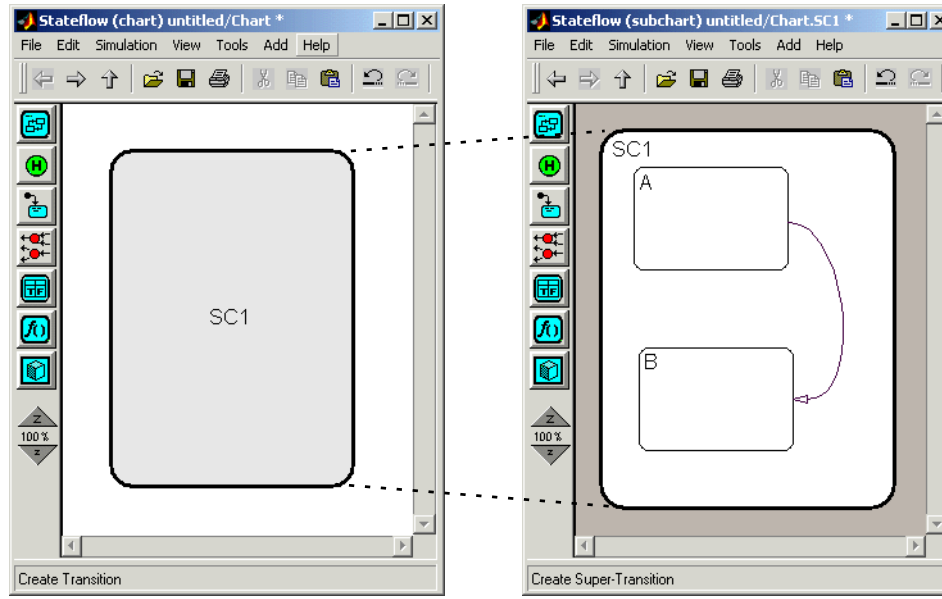
Caution You cannot undo the operation of converting a subchart back to its original form. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

Manipulating Subcharts as Objects

Subcharts also act as individual objects in Stateflow. You can move, copy, cut, paste, relabel, and resize subcharts as you would states and boxes. You can also draw transitions to and from a subchart and any other state or subchart at the same or different levels in the chart hierarchy (see "Using Supertransitions in Stateflow Charts" on page 5-71).

Opening a Subchart

Opening a subchart allows you to view and change its contents. To open a subchart, double-click anywhere in the block that represents the subchart. Stateflow replaces the current contents of the editor window with the contents of the subchart.



A shaded border surrounds the contents of the subchart. Stateflow uses the border to display supertransitions.

To return to the previous view, select **Back** from the Stateflow shortcut menu, press the **Esc** key on your keyboard, or select the up or back arrow on the Stateflow toolbar.

Editing a Subchart

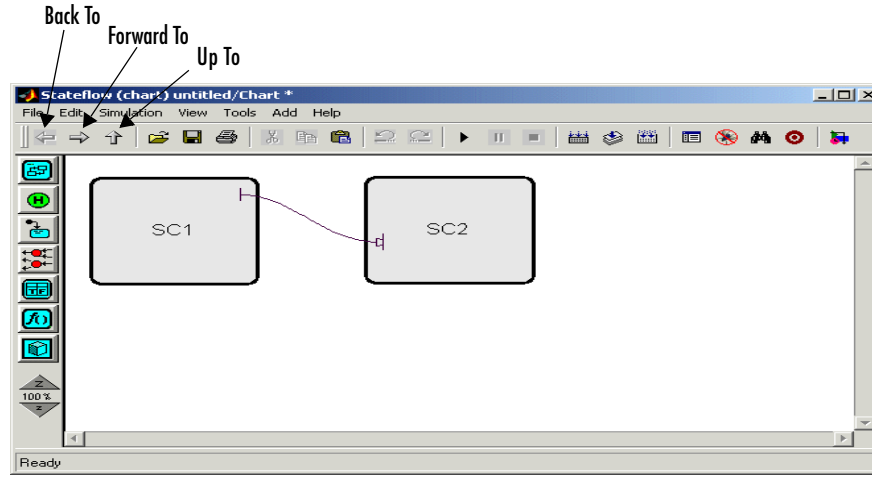
After you open a subchart (see "Opening a Subchart" on page 5-68), you can perform any editing operation on its contents that you can perform on a top-level chart. This means that you can create, copy, paste, cut, relabel, and resize the states, transitions, and subcharts in a subchart. You can also group states, boxes, and graphical functions inside subcharts.

You can also cut and paste objects between different levels in your chart. For example, to copy objects from a top-level chart to one of its subcharts, first open the top-level chart and copy the objects. Then open the subchart and paste the objects into the subchart.

Transitions from outside subcharts to states or junctions inside subcharts are called *supertransitions*. You create supertransitions differently than you do ordinary transitions. See "Using Supertransitions in Stateflow Charts" on page 5-71 for information on creating supertransitions.

Navigating Subcharts

The Stateflow toolbar contains a set of buttons for navigating a chart's subchart hierarchy.



- **Up To**

  If the Stateflow editor is displaying a subchart, this button replaces the subchart with the subchart's parent in the editor. If the editor is displaying a top-level chart, this button raises the Simulink model window containing that chart.

  The next two buttons allow you to retrace your steps as you navigate up and down a subchart hierarchy.

- **Back To**

  Returns to the chart that you visited before the current chart.

- **Forward To**

  Returns to the chart that you visited after visiting the current chart.

Using Supertransitions in Stateflow Charts

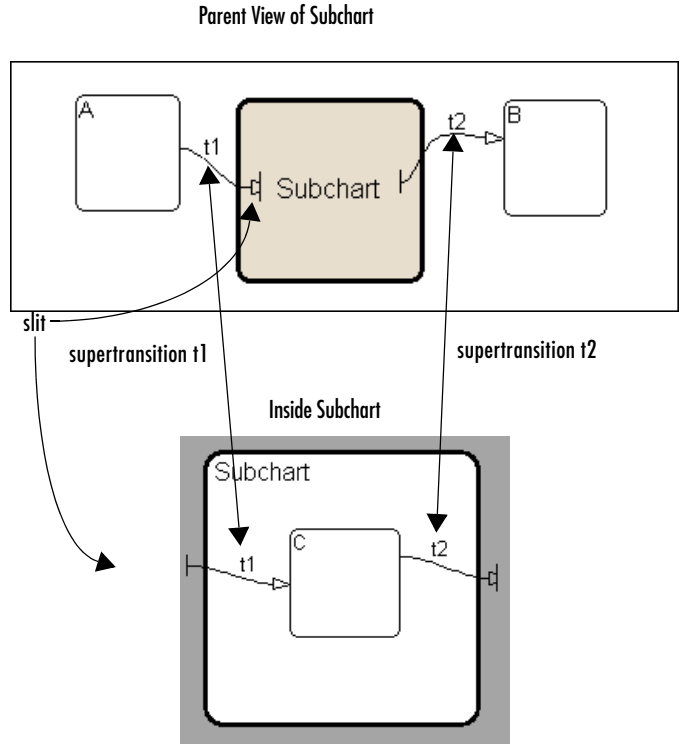
To connect transitions from outside a subchart to a state or junction inside a subchart, you'll need to know how to make a supertransition. Learn how to make supertransitions in the following topics:

- "What Is a Supertransition?" on page 5-71 — Describes supertransitions and gives you an example.
- "Drawing a Supertransition" on page 5-72 — Gives you the procedures for drawing supertransitions into and out of subcharts.
- "Labeling Supertransitions" on page 5-77 — Tells you how to label a supertransition composed of multiple transition segments.

What Is a Supertransition?

A *supertransition* is a transition between different levels in a chart, for example, between a state or junction in a top-level chart and a state or junction in one of its subcharts, or between states residing in different subcharts at the same or different levels in a diagram. Stateflow allows you to create supertransitions that span any number of levels in your chart, for example, from a junction at the top level to a state that resides in a subchart several layers deep in the chart.

The point where a supertransition enters or exits a subchart is called a *slit*. Slits divide a supertransition into graphical segments. For example, the following diagram shows two supertransitions as seen from the perspective of a subchart and its parent chart, respectively.



In this example, supertransition t1 goes from state A in the parent chart to state C in the subchart and supertransition t2 goes from state C in the subchart to state B in the parent chart. Note that both segments of t1 and t2 have the same label.

Drawing a Supertransition

The procedure for drawing a supertransition differs slightly depending on whether you are drawing the supertransition from an object outside the subchart to an object inside the subchart or whether you are drawing the supertransition from an object inside the subchart to an object outside the subchart. The following sections describe these two different procedures:

- "Drawing a Transition Into a Subchart" on page 5-73
- "Drawing a Transition Out of a Subchart" on page 5-75

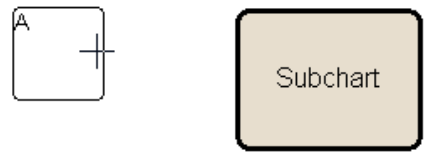
Caution You cannot undo the operation of drawing a supertransition. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

Drawing a Transition Into a Subchart

To draw a supertransition from an object outside a subchart to an object inside the subchart:

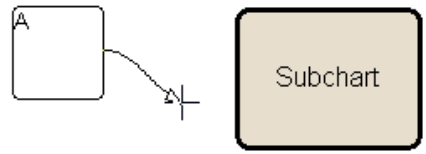
1 Position the mouse cursor over the border of the state.

The cursor assumes the crosshairs shape.



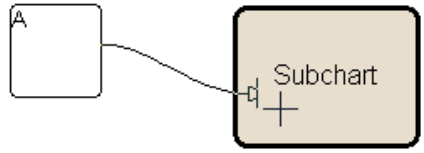
2 Drag the mouse.

Dragging the mouse causes a supertransition segment to appear. The segment looks like a regular transition. It is curved and is tipped by an arrowhead.



3 Drag the segment's tip anywhere just inside the border of the subchart.

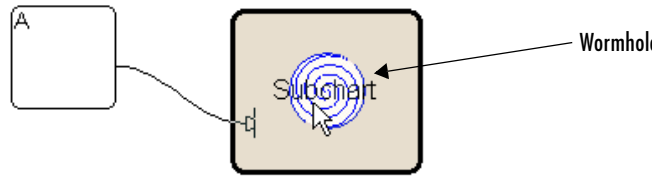
The arrowhead now penetrates the slit.



If you are not happy with the initial position of the slit, you can continue to drag the slit around the inside edge of the subchart to the desired location.

4 Continue dragging the cursor toward the center of the subchart.

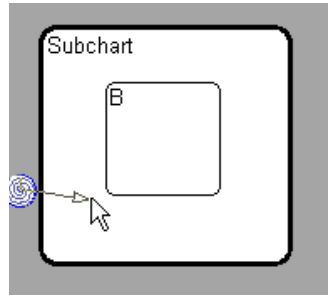
A wormhole appears in the center of the subchart.



A *wormhole* allows you to open a subchart while drawing a supertransition.

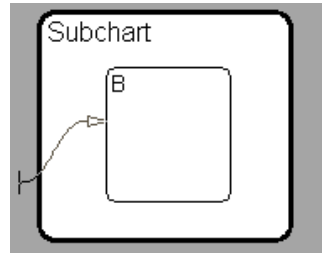
5 Drag the mouse pointer over the center of the wormhole.

The subchart opens. Now the wormhole and supertransition are visible inside the subchart.



6 Drag and drop the tip of the supertransition anywhere on the border of the object that you want to terminate the transition.

This completes the drawing of the supertransition.



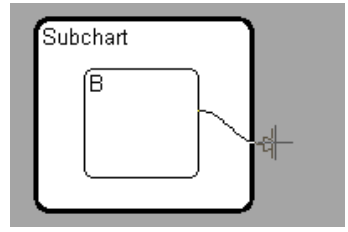
Note If the terminating object resides within a subchart in the current subchart, simply drag the tip of the supertransition through the wormhole of the inner subchart and complete the connection inside the inner chart. You can draw a supertransition to an object at any depth in the chart in this fashion.

Drawing a Transition Out of a Subchart

To draw a supertransition out of a subchart:

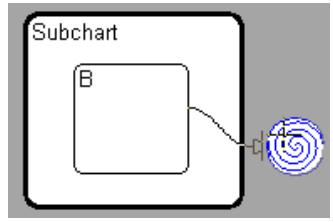
1 Draw an inner transition segment from the source object anywhere just outside the border of the subchart

A slit appears.



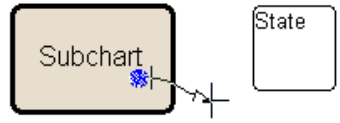
2 Keep dragging the transition away from the border of the subchart.

A wormhole appears.

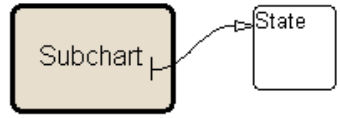


3 Drag the transition down the wormhole.

The parent of the subchart appears.



4 Complete the connection.



Note If the parent chart is itself a subchart and the terminating object resides at a higher level in the subchart hierarchy, you can continue drawing by dragging the supertransition into the border of the parent subchart. This allows you to continue drawing the supertransition at the higher level. In this way, you can connect objects separated by any number of layers in the subchart hierarchy.

Labeling Supertransitions

A supertransition is displayed with multiple resulting transition segments for each layer of containment traversed. For example, if you create a transition between a state outside a subchart and a state inside a subchart of that subchart, you create a supertransition with three segments, each displayed at a different containment level.

You can label any one of the transition segments constituting a supertransition using the same procedure used to label a regular transition (see "Labeling Transitions" on page 5-30). The resulting label appears on all the segments that constitute the supertransition. Also, if you change the label on any one of the segments, the change appears on all segments.

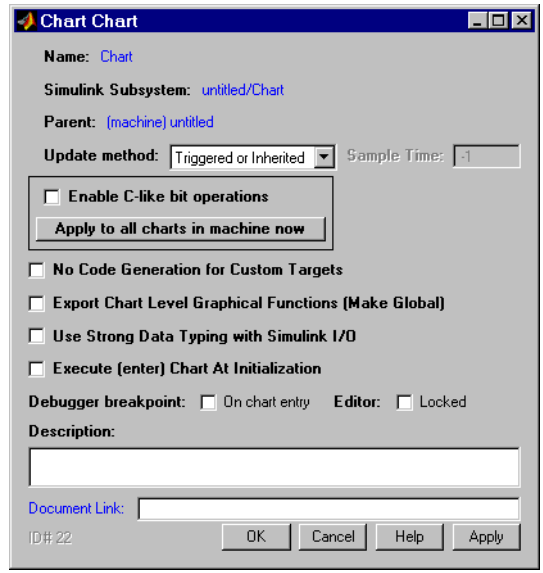
Specifying Chart Properties

Part of a chart's interface to its Simulink model is set when you specify the properties for a chart. To specify properties for the chart open in Stateflow's diagram editor, do the following:

1 Right-click an open area of the Stateflow diagram.

2 From the resultant context menu, select **Properties**.

The properties dialog for the chart appears as follows:



The following table lists and describes the chart object fields.

Field	Description
Name	Stateflow diagram name; read-only; click this hypertext link to bring the chart to the foreground.
Simulink Subsystem	Simulink subsystem name; read-only; click this hypertext link to bring the Simulink subsystem to the foreground.
Parent	Parent name; read-only; click this hypertext link to display the parent's property dialog box.
Update method	Stateflow lets you specify the method by which a simulation updates (wakes up) a chart in Simulink. Choose from **Triggered or Inherited**, **Sampled**, or **Continuous**. For more information, see "Setting the Stateflow Block Update Method" on page 8-4.
Sample Time	If **Update method** is **Sampled**, enter a sample time.

Field	Description (Continued)
Enable C-like bit operations	Select this box to recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode them as C bitwise operations. If this box is not selected, the following occurs: • `&` and `|` are interpreted as logical operators • `^` is interpreted as the power operator (for example, `2^3 = 8`). • The remaining expressions (>>, <<, and so on) result in parse errors. To specify this interpretation for all charts in the model (machine), select the **Apply to all charts in machine now** button.
No Code Generation for Custom Targets	Select this box to specify that no code is emitted when building for a custom target. This feature is useful for using Stateflow charts to simulate test environments during simulation testing before building to an actual embedded target.
Export Chart Level Graphical Functions	Exports graphical functions defined at the chart's root level. See "Exporting Graphical Functions" on page 5-53 for more information.

Field	Description (Continued)
Use Strong Data Typing with Simulink I/O	If this option is selected, the Chart block for this chart can accept and output signals of any data type supported by Simulink. The type of an input signal must match the type of the corresponding chart input data item (see "Defining Input Data" on page 6-26). Otherwise, a type mismatch error occurs. If this item is cleared, this chart accepts and outputs only signals of type `double`. In this case, Stateflow converts Simulink input signals to the data types of the corresponding chart input data items. Similarly, Stateflow converts chart output data (see "Defining Output Data" on page 6-27) to `double` if this option is not selected. For fixed-point data, see the note following this table.
Execute (enter) Chart at Initialization	Select this option if you want a chart's state configuration to be initialized at time 0 instead of at the first occurrence of an input event.
Debugger breakpoint	Select **On chart entry** to set a debugging breakpoint on entry to this chart.
Editor	Select **Locked** to mark the Stateflow diagram as read-only and prohibit any write operations.
Description	Textual description/comment.
Document Link	Enter a Web URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit/spec/data/speed.txt`.

Note For fixed-point data, the **Use Strong Data Typing with Simulink I/O** option is always on. Therefore, if an input or output fixed-point data in Stateflow does not match its counterpart data in Simulink, a mismatch error results.

3 Select one of the following buttons:

- **Apply** to save the changes
- **Cancel** to cancel any changes since the last apply
- **OK** to save the changes and close the dialog box
- **Help** to display the Stateflow online help in an HTML browser window

Checking the Chart for Errors

The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. Errors are displayed in informational pop-up windows. See “Parsing Stateflow Diagrams” on page 11-27 for more information.

Some aspects of the notation are verified at run-time. Others are verified during application run-time. Using the Debugger, you can detect the following run-time errors during simulation:

- State Inconsistency — Most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition. See “Debugging State Inconsistencies” on page 12-16.
- Transition Conflict — Occurs when there are two equally valid transition paths from the same source. See “Debugging Data Range Violations” on page 12-20.
- Data Range Violation — Occurs when minimum and maximum values specified for a data in its properties dialog are exceeded or when fixed-point data overflows its base word size. See “Debugging Data Range Violations” on page 12-20.
- Cyclical Behavior — Occurs when a step or sequence of steps repeats itself indefinitely. See “Debugging Cyclic Behavior” on page 12-22.

You can modify the notation to resolve run-time errors. See Chapter 12, “Debugging and Testing,” for more information on debugging run-time errors.

Creating Chart Libraries

A Stateflow chart library is a Simulink block library that contains Stateflow Chart blocks (and, optionally, other types of Simulink blocks as well). Just as Simulink libraries serve as repositories of commonly used blocks, chart libraries serve as repositories of commonly used charts.

You create a chart library in the same way you create other types of Simulink libraries. First, create an empty chart library by selecting **Library** from the **New** submenu of Simulink's **File** menu. Then create or copy Chart blocks into the library just as you would create or copy Chart blocks into a Stateflow model.

You use chart libraries in the same way you use other types of Simulink libraries. To include a chart from a library in your Stateflow model, copy or drag the chart from the library to the model. Simulink creates a link from the instance in your model to the instance in the library. This allows you to update all instances of the chart simply by updating the library instance.

Note Events parented by a library Stateflow machine are invalid. Stateflow allows you to define such events but flags them as errors when parsing a model.

Printing and Reporting on Charts

Stateflow offers the following reports for printing parts or all of your Stateflow chart:

- "Printing and Reporting on Stateflow Charts" on page 5-85— Tells you how to print the Simulink block diagram containing the Stateflow diagram currently displayed in the Stateflow diagram editor.
- "Generating a Model Report in Stateflow" on page 5-87 — Tells you how to make a comprehensive report of Stateflow objects relative to the currently displayed chart or subchart in the Stateflow diagram editor.
- "Printing the Current Stateflow Diagram" on page 5-90 — Tells you how to print the Stateflow diagram currently displayed in the Stateflow diagram editor.
- "Printing a Stateflow Book" on page 5-90 — Tells you how to generate a report that documents the Stateflow components of a Stateflow model.

You can also use the Report Generator for MATLAB and Simulink to generate a report that documents an entire Stateflow model, including both Simulink and Stateflow components. See the Report Generator for MATLAB documentation.

Printing and Reporting on Stateflow Charts

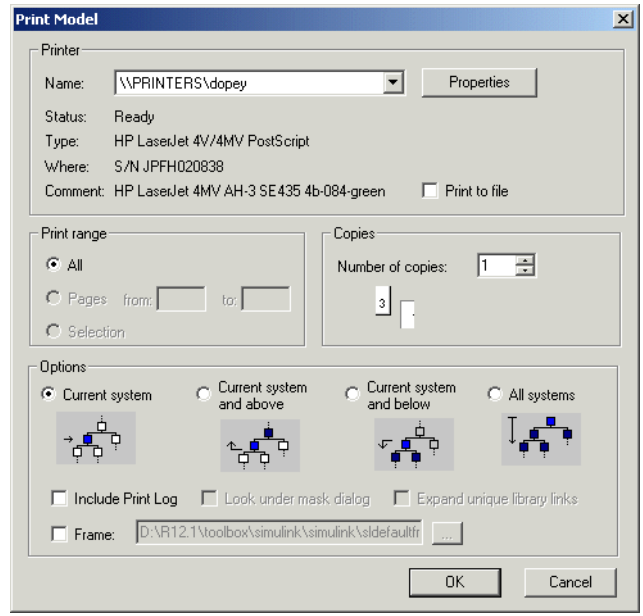
The **Print** option prints a copy of the current Stateflow diagram loaded in the Stateflow diagram editor. You can also select to print subcharts of the current diagram or the chart, subcharts, and Simulink subsystems that contain the current diagram.

Print a copy of a Stateflow diagram by doing the following:

1 Open the Stateflow chart or subchart you want to print.

2 Select **Print** from the **File** menu.

The **Print Model** dialog box appears as follows:



In the resulting **Print Model** window, select the printer for this report and one of the following options for the type of report you receive:

- **Current system** — Prints the current chart or subchart in view in the Stateflow editor.
- **Current system and above**— Prints the current chart or subchart in view in the Stateflow editor and all the subcharts and Simulink subsystems that contain it.
- **Current system and below**— Prints the current chart or subchart in view in the Stateflow editor and all the subcharts that it contains.
- **All systems** — Prints the current chart or subchart in view in the Stateflow editor, all the subcharts that it contains, and all the subcharts and Simulink subsystems that contain it.

Further enhance the above reports with the following options:

- **Include Print Log** — Includes a list of all printed diagrams as a preface to the print report.

- **Look under mask dialog** — Applies only to the masked subsystems that might appear in Simulink subsystems that are printed with the report options **Current system and below** and **All systems**.
- **Expand unique library links** — Applies only to the library blocks that might appear in Simulink subsystems that are printed with the report options **Current system and below** and **All systems**.
- **Frame** — Prints a title block frame that you specify in the adjacent field on each diagram in the report.

Note This option is also available in the Simulink window. See the topic "Printing a Block Diagram" in the Using Simulink documentation for more information on the preceding options and on the behavior of this command as used in Simulink. The information in this topic describes the behavior of this option only when it is used in a Stateflow diagram editor window.

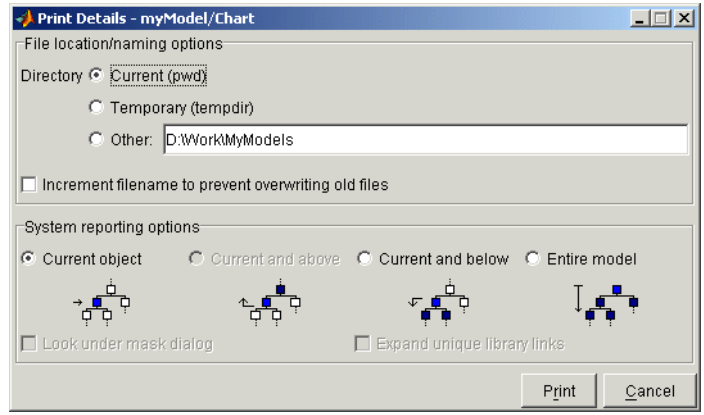
Generating a Model Report in Stateflow

The **Print Details** report in Stateflow is an extension to the **Print Details** report in Simulink. It provides a report of Stateflow and Simulink model objects relative to the Stateflow diagram currently in view in the Stateflow diagram editor from which you select the report.

To generate a model report on Stateflow diagram objects, do the following:

1. Open the Stateflow chart or subchart whose objects you want to report on.
2. In the diagram editor window, select **Print Details** from the **File** menu.

   The **Print Details** dialog box appears as follows:



3 Make selections for the destination directory of the report file and reporting options that determine what objects get reported.

   For details on setting the fields in the **File locations/naming options** section of this dialog, see "Generating a Model Report" in the Using Simulink documentation. For details on the report you receive from the report option you choose in the **System reporting options** section, see "System Report Options" on page 5-88 and "Report Format" on page 5-89.

4 Select **Print**.

The **Print Details** dialog box appears and tracks the activity of the report generator during report generation. See "Generating a Model Report" in the Using Simulink documentation for more details on this window.

If no serious errors are encountered, the resulting HTML report is displayed in your default browser.

System Report Options

Reports for the current Stateflow diagram vary with your choice of one of the **System reporting options** fields as follows:

- **Current** — Reports on the chart or subchart in the current Stateflow diagram editor and its immediate parent Simulink system.

- **Current and above** — This option is grayed out and unavailable for printing chart details in Stateflow.
- **Current and below** — Reports on the chart or subchart in the current Stateflow diagram editor and all contents at lower levels of containment (children) along with the immediate Simulink system.
- **Entire model** — Reports on the entire model including all Stateflow charts in the model for the chart in the current Stateflow diagram editor and all Simulink systems.

  If this option is selected, the following options are enabled to modify this report:

  - **Look under mask dialog** — Include the contents of masked subsystems in the report.
  - **Expand unique library links** — Include the contents of library blocks that are subsystems in the report.

    The report includes a library subsystem only once even if it occurs in more than one place in the model.

Report Format

The general top-down format of the **Print Details** report in Stateflow is as follows:

- The report is titled with the system in Simulink containing the chart or subchart in current view in Stateflow.
- A representation of Simulink hierarchy for the containing system and its subsystems follows. Each subsystem in the hierarchy is linked to the report of its Stateflow diagrams.
- The report section for the Stateflow diagrams of each system or subsystem begins with a small report on the system or subsystem and is followed by a report of each contained diagram.
- Each Stateflow diagram report includes a reproduction of its diagram with links for subcharted states that have reports of their own.
- Covered Stateflow and Simulink objects are tabulated and counted in a concluding appendix to the report.

Printing the Current Stateflow Diagram

The **Print Current View** option prints an individual Stateflow chart or subchart diagram as follows:

1. Open the chart or subchart that you want to print.
2. Select **Print Current View** from the Stateflow editor's **File** menu.
3. In the resulting submenu, choose one of the following destination options:

- **To File** — Converts the current view to a graphics file.

  Select the format of the graphics file from a resulting submenu of graphical file types.
- **To Clipboard** — Copies the current view to the system clipboard.

  Select the graphical format for the copy to the clipboard from a resulting submenu of graphical formats.
- **To Figure** — Converts the current view to a MATLAB figure window.
- **To Printer** — Prints the current view on the current printer.

You can also print the current view from the MATLAB command line using the `sfprint` command. See the "API Methods Reference" on page 16-1.

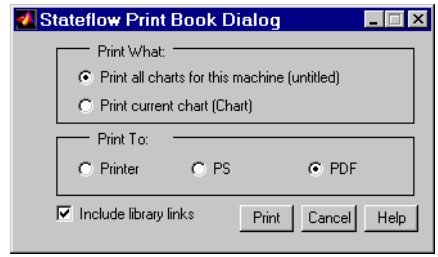
Printing a Stateflow Book

The **Print Book** report documents all the elements of a Stateflow chart, including states, transitions, junctions, events, and data. You can generate a book documenting a specific chart or all charts in a model.

To generate a book report of the objects of a Stateflow diagram, do the following:

1. Select and open a chart or subchart that you want to document.
2. Select **Print Book** from the Stateflow editor's **File** menu.

   The **Print Book** dialog box appears as follows:



3 Select the desired print options on the dialog.

4 Click the **Print** button to generate the report.

6

Defining Events and Data

Before you can use events and data in Stateflow, you must first define them. Learn how to define events and data in Stateflow in the following sections:

Defining Events (p. 6-2)	Learn how to define the events you need to trigger actions in Stateflow or its environment.
Defining Data (p. 6-14)	Learn how to define the data that Stateflow stores internally in its own workspace.

Defining Events

An event is a Stateflow object that triggers actions in a Stateflow machine or its environment. Stateflow defines a set of events that typically occur whenever a Stateflow machine executes (see "Implicit Events" on page 6-12). You can define other types of events that occur only during execution of a specific Stateflow machine or its environment.

This section contains the following topics:

- "Adding Events to the Data Dictionary" on page 6-2 — Tells you how to add events at any hierarchical object level within the Stateflow machine.
- "Setting Event Properties" on page 6-4 — Describes how to change the properties of an event in Stateflow.
- "Defining Local Events" on page 6-8 — Describes local events that are visible only to its parent object (for example, a state) and its children.
- "Defining Input Events" on page 6-8 — Describes input events that allow other Simulink blocks, including other Stateflow blocks, to notify a particular chart of events that occur outside it.
- "Defining Output Events" on page 6-9 — Describes output events that allow a chart to notify other blocks in a model of occurrences in that chart.
- "Exporting Events" on page 6-9 — Describes how to export events that enable external code to trigger events in the Stateflow machine.
- "Importing Events" on page 6-10 — Describes how to import events that allow a Stateflow machine built into a stand-alone or Real-Time Workshop target to trigger an event in external code.
- "Specifying Trigger Types" on page 6-11 — Describes the trigger type for a chart that defines how control signals trigger input and output events associated with the chart.
- "Implicit Events" on page 6-12 — Describes events that Stateflow triggers implicitly for actions such as entry in or exit from a state.

Adding Events to the Data Dictionary

You can use either the Stateflow diagram editor or Explorer to add up to 254 events. If you add events in the Stateflow diagram editor, they are visible to all objects in the chart. You must use the Stateflow Explorer to add events that are visible only within the state, chart, or Stateflow machine that you add them to.

Using the Stateflow Editor

To use the Stateflow editor to add an event, do the following:

1 From the **Add** menu of the Stateflow editor, select **Event**.

2 In the resulting submenu, select the event's scope (see "Scope" on page 6-6 for an explanation).

Stateflow adds a default definition of the new event to the Stateflow data dictionary and displays the **Event** dialog box.

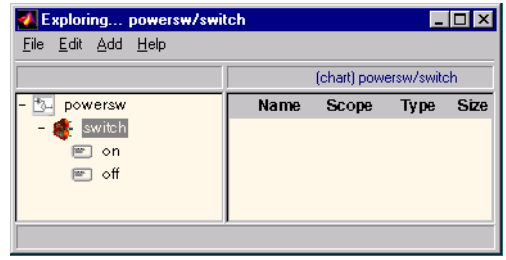
3 Use the **Event** dialog box to specify event options (see "Setting Event Properties" on page 6-4).

Using the Explorer to Define Events

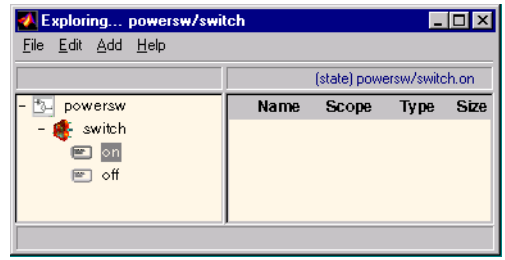
To use the Stateflow Explorer to define an event:

1 Select **Explore** from the Stateflow editor's **Tools** menu.

Stateflow opens the Explorer.

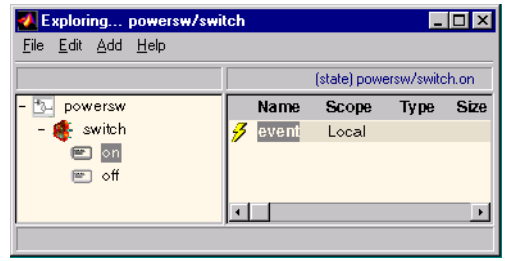


2 Select the object (machine, chart, or state) in the Explorer's object hierarchy pane where you want the new event to be visible.



3 Select **Event** from the Explorer's **Add** menu.

Stateflow adds a default definition for the new event in the data dictionary and displays an entry for the new event in the Explorer's content pane.



4 Set the new event's properties to values that reflect its intended usage (see "Setting Event Properties" on page 6-4).

Setting Event Properties

To change the properties for a data object in Stateflow, access its **Event** dialog as follows:

1 Select **Explore** from the Stateflow editor's **Tools** menu.

2 In the resulting Stateflow Explorer window, expand sections of the hierarchy to find the event whose properties you want to change.

3 Right-click the event in the Explorer's hierarchy pane (left pane).

4 Select **Properties** from the resulting context menu.

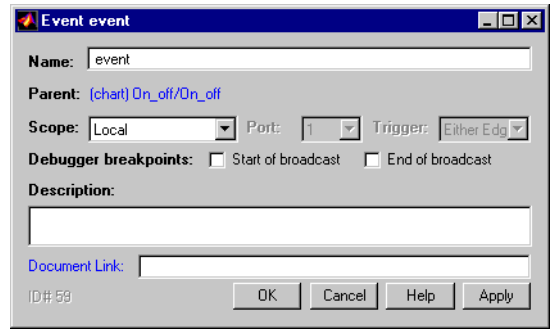
   In place of steps 3 and 4 you could also simply select the event and then select **Properties** from the **Edit** menu.

5 In the resulting **Event** dialog, set the item's properties.

6 Select **OK** to apply your changes and dismiss the **Event** dialog.

Note You can also set an event's **Scope** (see "Defining Local Events" on page 6-8) and **Trigger** properties by editing the corresponding fields in the event's entry in the Explorer's contents pane instead of opening its **Event** dialog.

The **Event** dialog box allows you to specify event properties.



The dialog box displays the following fields and options.

Name

Name of this event. Event names enable actions to reference specific events. You assign a name to an event by setting its `Name` property. You can assign any name that begins with an alphabetic character, does not include spaces, and is not shared by sibling events.

Parent

Clicking this field displays the parent of this event in the Stateflow editor. The parent is the object in which this event is visible. When an event is triggered, Stateflow broadcasts the event to the parent and all the parent's descendants. An event's parent can be a Stateflow machine, a chart, or a state. You specify an event's parent when you add it to the data dictionary (see "Adding Events to the Data Dictionary" on page 6-2).

Scope

Scope of this event. The scope specifies where the event occurs relative to its parent. The following sections describe each scope setting.

Local. This event occurs in a Stateflow machine and is parented by the machine or one of its charts or states. See "Defining Local Events" on page 6-8 for more information.

Input from Simulink. This event occurs in one Simulink block and is broadcast in another. The first block can be any type of Simulink block. The second block must be a Chart block. See "Defining Input Events" on page 6-8 for more information.

Output to Simulink. This event occurs in one Simulink block and is broadcast in another. The first block is a Chart block. The second block can be any type of Simulink block. See "Defining Output Events" on page 6-9 for more information.

Exported. An exported event is a Stateflow event that can be broadcast by external code built into a stand-alone or Real-Time Workshop target. See "Exporting Events" on page 6-9 for more information.

Imported. An imported event is an externally defined event that can be broadcast by a Stateflow machine embedded in the external code. See "Importing Events" on page 6-10 for more information.

Note If you copy a Stateflow Chart block from one Simulink model to another, all objects in the chart hierarchy are copied except those parented by the Stateflow machine. This means that events parented by the original Stateflow machine (local, imported, exported) are not copied to the new machine. You can, however, use the Explorer tool to copy individual events by click-dragging them from machine to machine.

Trigger

Type of signal that triggers an input or output event. See "Specifying Trigger Types" on page 6-11 for more information.

Index

Index of the input signal that triggers this event. This option applies only to input events and appears when you select `Input from Simulink` as the scope of this event. See "Associating Input Events with Control Signals" on page 6-8 for more information.

Port

Index of the port that outputs this event. This property applies only to output events and appears when you select `Output to Simulink` as the scope of this event. See "Associating an Output Event with an Output Port" on page 6-9 for more information.

Description

Description of this event. Stateflow allows you to store brief descriptions of events in the data dictionary. To describe a particular event, enter its description in this field.

Document Link

Stateflow allows you to provide online documentation for events defined by a model. To document a particular event, set its `Documentation` property to a MATLAB expression that displays documentation in some suitable online format (for example, an HTML file or text in the MATLAB command window). Stateflow evaluates the expression when you click the event's documentation

link (the blue text that reads "Document Link" displayed at the bottom of the event's **Event** dialog box).

Defining Local Events

A local event is an event that can occur anywhere in a Stateflow machine but is visible only in its parent (and its parent's descendants). To define an event as local, set its `Scope` property to `Local`.

Defining Input Events

An input event occurs outside a chart and is visible only in that chart. This type of event allows other Simulink blocks, including other Stateflow blocks, to notify a particular chart of events that occur outside it. To define an event as an input event, set its `Scope` property to `Input from Simulink`.

You can define multiple input events for a chart. The first time you define an input event for a chart, Stateflow adds a trigger port to the chart's block. External blocks can trigger the chart's input events via a signal or vector of signals connected to the chart's trigger port by associating input events with control signals. When defining input events for a chart, you must specify how control signals connected to the chart trigger the input events (see "Specifying Trigger Types" on page 6-11).

You can also add input events to a chart from Simulink in the Stateflow Explorer. See "Adding Input Events from Simulink" on page 8-6.

Associating Input Events with Control Signals

An input event's `Index` property associates the event with a specific element of a control signal vector connected to the trigger port of the chart that parents the event. The first element of the signal vector triggers the input event whose index is 1; the second, the event whose index is 2; and so on. Stateflow assigns 1 as the index of the first input event that you define for a chart, 2 as the index of the second event, and so on. You can change the default association for an event by setting the event's `Index` property to the index of the signal that you want to trigger the event.

Input events occur in ascending order of their indexes when more than one such event occurs during update of a chart (see "Setting the Stateflow Block Update Method" on page 8-4). For example, suppose that when defining input events for a chart, you assign the indexes 3, 2, and 1 to events named A, B, and

C, respectively. Now, suppose that, during simulation of the model containing the chart, that events A and C occur in a particular update. Then, in this case, the order of occurrence of the events is C first followed by A.

Defining Output Events

An output event is an event that occurs in a specific chart and is visible in specific blocks outside the chart. This type of event allows a chart to notify other blocks in a model of events that occur in the chart. To define an event as an output event, set its `Scope` property to `Output to Simulink`. You can define multiple output events for a given chart. Stateflow creates a chart output port for each output event that you define (see "Port" on page 6-7). Your model can use the output ports to trigger the output events in other Simulink blocks in the same model.

You can also add output events to a chart from Simulink in the Stateflow Explorer. See "Adding Output Events to Simulink" on page 8-7.

Associating an Output Event with an Output Port

An output event's `Port` property associates the event with an output port on the chart block that parents the event. The property specifies the position of the port relative to other event ports on the Chart block. Event ports appear below data ports on the right side of a chart block. Stateflow numbers ports sequentially from top to bottom, starting with port 1. Stateflow assigns port 1 to the first output event that you define for a chart, port 2 to the second output event, and so on. You can change the default port assignment of an event by resetting its `Port` property or by selecting the output event in the Explorer and dragging and dropping it to the desired position in the list of output events.

Exporting Events

Stateflow allows a Stateflow machine to export events. Exporting events enables external code to trigger events in the Stateflow machine. To export an event, first add the event to the data dictionary as a child of the Stateflow machine (see "Adding Events to the Data Dictionary" on page 6-2). Then set the new event's `Scope` property to `Exported`.

Note External events can be parented only by a Stateflow machine. This means that you must use the Explorer to add external events to the data dictionary. It also means that external events are visible everywhere in the Stateflow machine.

The Stateflow code generator generates a function for each exported event. The C prototype for the exported event function has the form

```
void external_broadcast_EVENT()
```

where `EVENT` is the name of the exported event. External code built into a target can trigger the event by invoking the event function. For example, suppose you define an exported event named `switch_on`. External code can trigger this event by invoking the generated function `external_broadcast_trigger_on`. See "Exported Events" on page 8-23 for an example of how to trigger an exported event.

Importing Events

The Stateflow machine can import events defined by external code. Importing an event allows the Stateflow machine built into a stand-alone or Real-Time Workshop target to trigger the event in external code. To import an event, first add the event to the data dictionary as a child of the Stateflow machine that needs to trigger the event (see "Adding Events to the Data Dictionary" on page 6-2). Then set the new event's `Scope` property to `Imported`.

Note The Stateflow machine serves as a surrogate parent for imported events. This means that you must use the Explorer to add imported events to the data dictionary.

Stateflow assumes that external code defines each imported event as a function whose prototype is of the form

```
void external_broadcast_EVENT
```

where `EVENT` is the Stateflow name of the imported event. For example, suppose that the Stateflow machine imports an external event named

`switch_on`. Then Stateflow assumes that external code defines a function named `external_broadcast_switch_on` that broadcasts the event to external code. When building a target for the Stateflow machine, the Stateflow code generator encodes actions that signal imported events as calls to the corresponding external broadcast event functions defined by the external code.

Specifying Trigger Types

A trigger type defines how control signals trigger input and output events associated with a chart. Trigger types fall into two categories: function call and edge. The basic difference between these two types is when receiving blocks are notified of their occurrence. Receiving blocks are notified of edge-triggered events only at the beginning of the next simulation time step, regardless of when the events occurred during the previous time step. By contrast, receiving blocks are notified of function-call-triggered events the moment the events occur, even if they occur in mid step.

You set a chart's trigger type by setting the `Trigger` property of any of the input or output events defined for the chart. If you want a chart to notify other blocks the moment an output event occurs, set the `Trigger` property of the output event to `Function Call`. The output event's trigger type must be `Either Edge`. If a chart is connected to a block that outputs function-call events, you must specify the `Trigger` property of the receiving chart's input events as `Function Call`. Stateflow changes all the chart's other input events to `Function Call`.

If it is not critical that blocks be notified of events the moment they occur, you can define the events as edge-triggered. You can specify any of the falling types of edge triggers.

- **Rising Edge** — A rising level on the control signal triggers the corresponding event.
- **Falling Edge** — A falling level on the control signal triggers the event.
- **Either Edge** — A change in the signal level triggers the event.

In all cases, the signal must cross 0 to constitute a valid trigger. For example, a change from -1 to 1 constitutes a valid rising edge, but not a change from 1 to 2.

If you specify an edge trigger type that differs from the edge type previously defined for a chart, Stateflow changes the **Trigger** type of the chart's input events to **Either Edge**.

Implicit Events

Stateflow defines and triggers the following events that typically occur whenever a chart executes:

- Chart waking up
- Entry into a state
- Exit from a state
- Value assigned to an internal (noninput) data object

These events are called *implicit events* because you do not have to define or trigger them explicitly. Implicit events are children of the chart in which they occur. Thus, they are visible only in the charts in which they occur.

Referencing Implicit Events

Action expressions can use the following syntax to reference implicit events.

```
event(object)
```

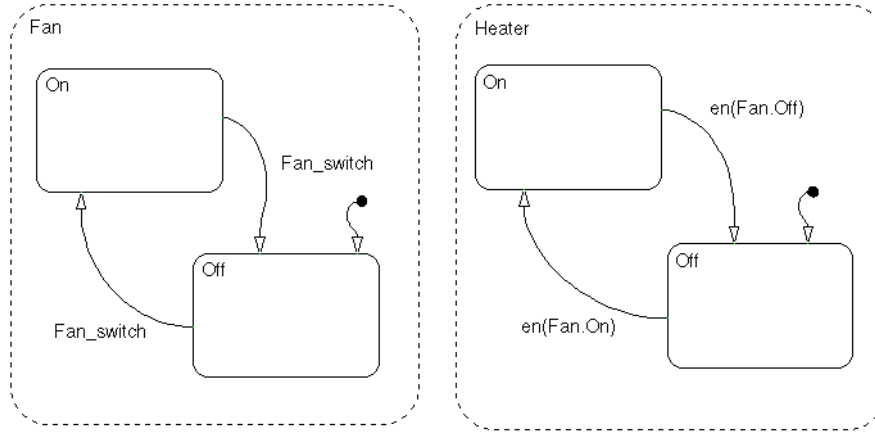
where `event` is the name of the implicit event and `object` is the state or data in which the event occurred. Valid implicit event names (and their shortcuts) are `wakeup` (also referred to as `tick`), `enter` (`en`), `exit` (`ex`), and `change` (`chg`).

If more than one object has the same name, the event reference must qualify the object's name with that of its ancestor. The following are some examples of valid implicit event references.

```
enter(switch_on)
en(switch_on)
change(engine.rpm)
```

Note The `wakeup` (or `tick`) event always refers to the chart of the action being evaluated. It cannot reference a different chart by argument.

This example illustrates use of an implicit `enter` event.



`Fan` and `Heater` are parallel (AND) superstates. By default, the first time the Stateflow diagram is awakened by an event, the states `Fan.Off` and `Heater.Off` become active. The first time event `Fan_switch` occurs, the transition from `Fan.Off` to `Fan.On` occurs. When `Fan.On`'s entry action executes, an implicit local event is broadcast (i.e., `en(Fan.On) == 1`). This event broadcast triggers the transition from `Heater.Off` to `Heater.On` (triggered by the condition `en(Fan.On)`. Similarly, when the system transitions from `Fan.On` to `Fan.Off` and the implicit local event `Fan.Off` is broadcast, the transition from `Heater.On` to `Heater.Off` is triggered.

Defining Data

Stateflow can store and retrieve data that resides internally in its own workspace. It can also access data that resides externally in the Simulink model or application that embeds the Stateflow chart. When creating a Stateflow model, define any internal or external data referenced by Stateflow actions as described in the following topics:

- "Adding Data to the Data Dictionary" on page 6-14 — Describes how to add a data object to any object in a Stateflow chart, to the chart itself, or to the Stateflow machine.
- "Setting Data Properties" on page 6-16 — Describes how to change the properties for a data object in Stateflow.
- "Defining Data Arrays" on page 6-24 — Describes how to specify a data object as an array in its properties dialog.
- "Defining Input Data" on page 6-26 — Describes how to specify input data that supply data to a chart from Simulink using chart input ports.
- "Defining Output Data" on page 6-27 — Describes how to specify output data that a Stateflow chart supplies to other blocks in Simulink using chart output ports.
- "Associating Ports with Data" on page 6-27 — Describes how Stateflow associates the first input port with the first input item you define, the first output port with the first output item, the second input port with the second input item, and so on.
- "Defining Temporary Data" on page 6-28 — Describes temporary data that you define by setting the data's **Scope** property to Temporary.
- "Exporting Data" on page 6-28 — Describes exported data that enables external code and any object in the Stateflow machine to access the data.
- "Importing Data" on page 6-29 — Describes imported data that allows a chart in the Stateflow machine to import definitions of data defined by external code that embeds the Stateflow machine.

Adding Data to the Data Dictionary

You can use either the Stateflow editor or Explorer to add data that is accessible only in a specific chart. You must use the Stateflow Explorer to add

data that is accessible everywhere in a Stateflow chart or in a specific object such as a state.

Using the Stateflow Editor to Add Data

To use the Stateflow editor to add data, do the following:

1. From the **Add** menu of the Stateflow editor, select **Data**.
2. On the resulting submenu, select the data's scope (see "Scope" on page 6-18).

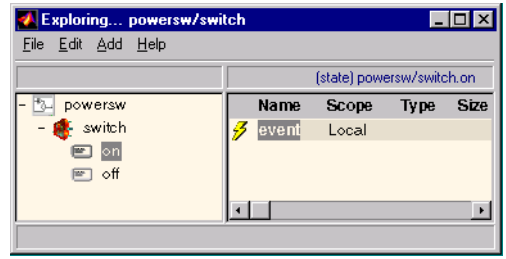
   Stateflow adds a default definition of the new item to the Stateflow data dictionary and displays a **Data** dialog that displays the new item's default properties.
3. Use the **Data** dialog box to set the new item's properties to reflect its intended usage (see "Setting Data Properties" on page 6-16).

Using the Explorer to Add Data

To use the Stateflow Explorer to define a data item:

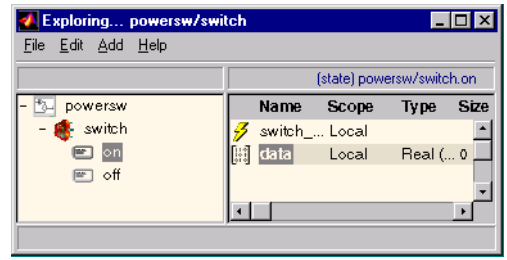
1. Select **Explore** from the Stateflow editor's **Tools** menu.

   Stateflow opens the Explorer.



2. Select the object (machine, chart, or state) in the Explorer's object hierarchy pane where you want the new item to be accessible.
3. Select **Data** from the Explorer's **Add** menu.

   Stateflow adds a default definition for the new item in the data dictionary and displays an entry for the item in the Explorer's content pane.



4 Set the new item's properties to values that reflect its intended usage (see "Setting Event Properties" on page 6-4).

Setting Data Properties

To change the properties for a data object in Stateflow, access its **Data** dialog as follows:

1 Select **Explore** from the Stateflow editor's **Tools** menu.

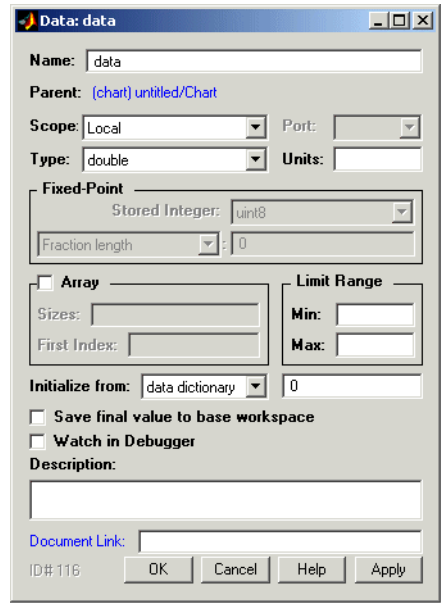
2 In the resulting Stateflow Explorer window, expand sections of the hierarchy to find the data whose properties you want to change.

3 Right-click the data object in the Explorer's hierarchy pane (left pane).

4 Select **Properties** from the resulting context menu.

In place of steps 3 and 4 you could also simply select the data object and then select **Properties** from the **Edit** menu.

5 In the resulting **Data** dialog, set the item's properties.



Note You can also set some of the data's properties by editing the corresponding column fields in the item's entry in the Explorer's **Contents** pane instead of opening the **Data** dialog for the data object.

---

6 Click **OK** to apply your changes and dismiss the **Data** dialog box.

The **Data** dialog box includes the following settings:

Name

Name of the data item. A data name can be of any length and can consist of any alphanumeric and special character combination, with the exception of embedded spaces. The name cannot begin with a numeric character.

Parent

Parent of this data item. The parent determines the objects that can access it. Specifically, only the item's parent and descendants of that parent can access

the item. You specify the parent of a data item when you add the item to the data dictionary.

Scope

Scope of this data item. A data object's scope specifies where it resides in memory relative to its parent. These are the options for the **Scope** property:

Local. A local data object resides and is accessible only in a machine, chart, or state.

Input from Simulink. This is a data item that is accessible in a Simulink Chart block but resides in another Simulink block that might or might not be a Chart block. The receiving Chart block reads the value of the data item from an input port associated with the data item. See "Importing Data" on page 6-29 for more information.

Output to Simulink. This is a data item that resides in a Chart block and is accessible in another block that might or might not be a Chart block. The Chart block outputs the value of the data to an output port associated with the data item. See "Defining Output Data" on page 6-27 for more information.

Temporary. A temporary data item exists only while its parent is executing. See "Defining Temporary Data" on page 6-28 for more information.

Constant. A constant data object is read-only and retains the initial value set in its **Data** properties dialog box.

Exported. An exported data item is data assigned to a Stateflow machine that can be accessed by external code that embeds that machine. See "Exporting Data" on page 6-28 for more information.

Imported. Imported data is data defined by external code that can be accessed by a Stateflow machine embedded in the external code. See "Importing Data" on page 6-29 for more information.

Input. Applies to a graphical function (arguments). See "Creating a Graphical Function" on page 5-48 for more information.

Output. Applies to a graphical function (return data). See "Creating a Graphical Function" on page 5-48 for more information.

Note If you copy a Stateflow Chart block from one Simulink model to another, all objects in the chart hierarchy are copied except those parented by the Stateflow machine. This means that data parented by the original Stateflow machine (local, imported, exported) is not copied to the new machine. You can, however, use the Explorer tool to copy individual data by click-dragging it from machine to machine.

Type

Data type of this data. Select one of the following:

Type	Description
`double`	Double-precision floating-point (64 bit)
`single`	Single-precision floating-point (32 bit)
`int32`	32 bit signed integer
`int16`	16 bit signed integer
`int8`	8 bit signed integer
`uint32`	32 bit unsigned integer
`uint16`	16 bit unsigned integer
`uint8`	8 bit unsigned integer
`boolean`	Boolean

Type	Description
`fixpt`	Stateflow's fixed-point data type. Specify the fixed-point type through the **Stored Integer** and **Scaling** fields in the adjacent **Fixed-Point** field section, which is enabled when you specify the `fixpt` type. See "Fixed-Point Data" on page 7-14 for a description of the Stateflow fixed-point type and its use in Stateflow. For fixed-point data, the **Use Strong Data Typing with Simulink IO** option in the chart properties dialog (see "Specifying Chart Properties" on page 5-78) is always on. This means that if input or output fixed-point data in Stateflow does not match its counterpart data in Simulink, a mismatch error results.
`ml`	Used for holding MATLAB data in Stateflow. See "ml Data Type" on page 7-51.

Fixed-Point

You select an integer base for the fixed-point data in the **Stored Integer** field. The scaling type for the fixed-point number is entered in the drop-down field below which can take one of the two selectable values, **Fraction Length** or **Scaling**. The scaling value is entered in the field to the right. For a detailed description of fixed-point data, see "Fixed-Point Data" on page 7-14. See also "Specifying Fixed-Point Data in Stateflow" on page 7-19.

Stored Integer. Select the integer word type to hold the fixed-point data quantized integer. The size of the word in bits is indicated by a suffix of 8, 16, or 32. Larger word sizes can represent large quantities with greater precision than smaller word sizes. Unsigned types (uint8, uint16, uint32) can represent only positive quantities. Signed types (int8, int16, int32) can represent negative quantities.

Fraction length. Select this value to interpret the entry in the right adjacent field as the binary point location for binary-point-only scaling. A positive integer entry moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the

right. A negative integer entry moves the binary point further right of the rightmost bit by that amount. Only integers can be entered (see note below).

Scaling. Select this value to interpret the entry in the right adjacent field as scaling with separate slope and bias entered in [Slope Bias] format (can be entered with or without brackets `[]`). For example, the entry `[2 3]` sets the slope to 2 and the bias to 3. You can also enter the slope or bias as an arithmetic expression, for example, `[1.2*2^-2 3.4]`. If you enter a single value, it is interpreted as the slope. The value entered for the slope can be any positive real number. The value for the bias can be any real number. See following note.

Note Values entered for **Fraction length** or **Scaling** that exceed the ranges specified in the previous sections are flagged immediately with red text.

Port

Index of the port associated with this data item (see "Associating Ports with Data" on page 6-27). This control applies only to input and output data.

Units

Units, for example, inches, centimeters, and so on, represented by this data item. The value of this field is stored with the item in the Stateflow machine's data dictionary.

Array

All data that you add is scalar in the specified **Type** (see preceding) by default. If you select the **Array** check box, this data item is an array. However, the Scope value that you assign this data limits the kind of array this data can be. See the following **Sizes** property.

When you select the **Array** check box, the **Sizes** and **First Index** fields are enabled in the **Data** dialog.

Sizes. Size of this array. The value of this property can be a scalar or a MATLAB vector. If it is a scalar (for example, `5`), it specifies the size of a one-dimensional array (that is, a vector). If it is a MATLAB vector (for example, '[2 3 7]'), it indicates the size of each dimension of a multidimensional array whose number of dimensions corresponds to the length of the vector.

The allowed size of the data array that you specify depends on the data's **Scope** property, as follows:

Scope	Scalar	Vector	Matrix 2-dim	Matrix n-dim
Constant	Yes	No	No	No
Input from Simulink/Output to Simulink	Yes	Yes	Yes	No
Temporary/Local	Yes	Yes	Yes	Yes
Imported/Exported	Yes	Yes	Yes	Yes
Input (to function)/ Output (to function)	Yes	No	No	No

First Index. Specifies the index of the first element of this array. For example, the first index of a zero-based array is `0`.

Limit Range

This control group specifies values used by a Stateflow machine to check the validity of this data item. It includes the next two controls.

Min. Minimum value that this data item can have during execution or simulation of the Stateflow machine it belongs to.

Max. Maximum value that this data item can have during execution or simulation of the Stateflow machine it belongs to.

Initialize from

Source of the initial value for this data item: either the Stateflow data dictionary or the MATLAB workspace. If this data item is an array, Stateflow sets each element of the array to the specified initial value.

If the source is the data dictionary, enter the initial value in the adjacent text field. Stateflow stores the value that you enter in the data dictionary.

If the source is the MATLAB workspace, this item gets its initial value from a similarly named variable in the MATLAB workspace. For example, suppose

that the name of a Stateflow data item is A. If the parent workspace has a variable named A, this value is used to initialize the data item.

The following table summarizes the time of initialization for each data scope.

Data Parent	Scope	When Initialized
Machine	Local	Start of simulation
	Exported	Start of simulation
	Imported	Not applicable
Chart	Input	Not applicable
	Output Local	Start of simulation or when chart is reinitialized as part of an enabled Simulink subsystem
State with History junction	Local	Start of simulation or when chart is reinitialized as part of an enabled Simulink subsystem
State without History junction	Local	State activation (state entered)
Graphical Function	Input	Not applicable
	Output	When function is invoked
	Local	Start of simulation or when chart is reinitialized as part of an enabled Simulink subsystem

Note You can also use the Stateflow Explorer to set the **Initialize from** field.

Save final value to base workspace

Selecting this option causes the value of the data item to be assigned to a similarly named variable in the model's base workspace at the end of simulation.

Watch in debugger

If selected, this option causes the debugger to halt if this data item is modified.

Description

Description of this data item.

Document link

Stateflow allows you to provide online documentation for data defined by a model. To document a particular item of data, set this property to a MATLAB expression that displays documentation in some suitable online format (for example, an HTML file or text in the MATLAB Command Window). Stateflow evaluates the expression when you click the item's documentation link (the blue text that reads `Document Link` displayed at the bottom of the event's **Data** dialog box).

Defining Data Arrays

To add a data array, do the following:

1. Add a default data item to the data dictionary as a child of the state, chart, or machine that needs to access the data (see "Adding Data to the Data Dictionary" on page 6-14).
2. Open the **Data** dialog box.
3. Select the **Array** check box on the dialog.
4. Set the data's **Sizes** property to the size of each of the array's dimensions. See the **Sizes** property in "Setting Data Properties" on page 6-16.

   For example, to define a 100-element vector, set the **Sizes** property to `100`. To define a 2-by-4 array, set the **Sizes** property to `[2 4]`.

5 Set the item's **Initial Index** property to the index of the array's first element.

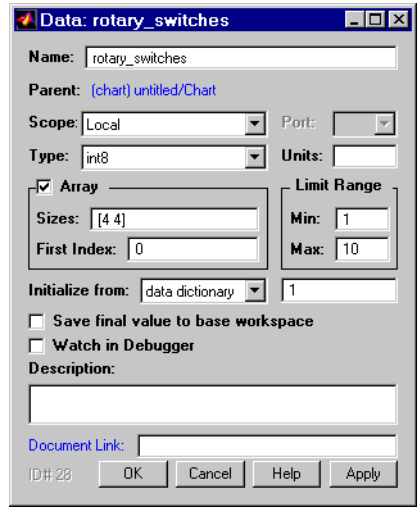
   For example, to define a zero-based array, set the **Initial Index** property to `0`.

6 Set the item's initialization source and, if initialized from the data dictionary, initial value.

   For example, to specify that an array's elements be initialized to zero, set the **Initialize from** option in the **Data** dialog box to `data dictionary` and enter `0` in the adjacent text field.

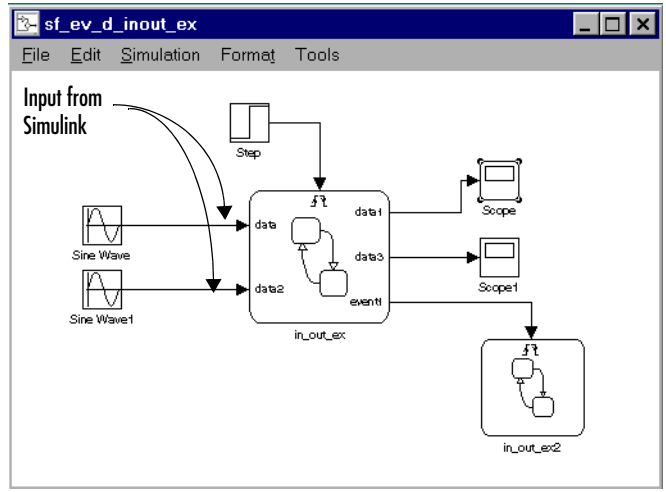
7 Set the other options in the dialog box (for example, **Name**, **Type**, and so on) to reflect the data item's intended usage.

Suppose that you want to define a local, 4-by-4, zero-based array of type `Integer` named `rotary_switches`. Further, suppose that each element of the array is initially 1 and can have no values less than 1 or greater than 10. The following **Data** dialog box shows the settings for such an array.



Defining Input Data

Stateflow allows a model to supply data to a chart via input ports on the chart's block. Such data is called input data. To define an item of input data, add a default item to the Stateflow data dictionary as a child of the chart that will input the data. Set the new item's scope to **Input from Simulink**. Stateflow adds an input port to a chart for each item of input data that you define for the chart.

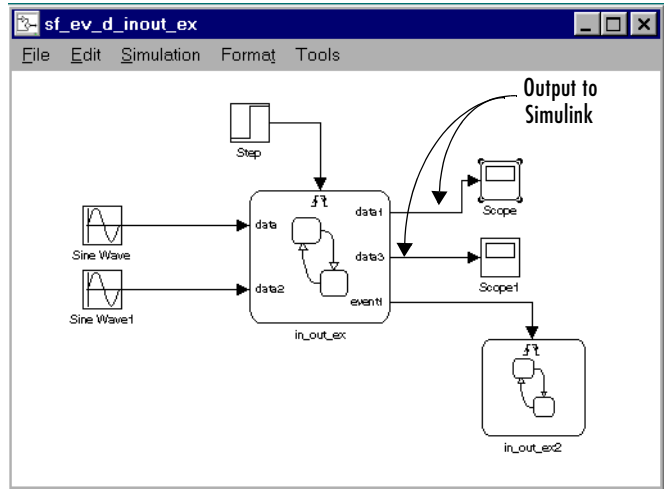


Set the item's other properties (for example, **Name**, **Type**, and so on) to appropriate values.

You can set an input item's data type to any Stateflow-supported type. You can set the size of the input data to be scalar, vector, or two-dimensional array (see "Defining Data Arrays" on page 6-24). If the chart's strong data typing option is enabled (see "Specifying Chart Properties" on page 5-78), input signals must match the specified type. Otherwise, a mismatch error occurs. If strong data typing is not enabled, input signals must be of type `double`. In this case, Stateflow converts the input value to the specified type. If the input item is a vector, the model must supply the data via a signal vector connected to the corresponding input port on the chart.

Defining Output Data

Output data is data that a chart supplies to other blocks via its output ports. To define an item of output data, add a default data item to the data dictionary as a child of the chart that supplies the item. Then, set the new item's **Scope** property to `Output to Simulink`. Stateflow adds an output port to the chart for each item that it outputs.



You can set an output item's type to any supported Stateflow data type. You can set the size of the input data to be scalar, vector, or two-dimensional array (see "Defining Data Arrays" on page 6-24). If the chart's strong data typing option is enabled (see "Specifying Chart Properties" on page 5-78), the chart outputs a Simulink signal of the same data type as the output data item's type. If the option is not enabled, the Stateflow Chart block converts the output data to Simulink type `double`.

Associating Ports with Data

Stateflow creates and associates an input port with each input data item that you define for a chart and an output port for each output data item. By default, Stateflow associates the first input port with the first input item you define, the first output port with the first output item, the second input port with the second input item, and so on. The **Data** dialog for each item shows its current port assignment in the **Port** field. You can alter the assignment by editing the

value displayed in the **Port** field or by selecting the data item in the Explorer and dragging it to the desired location in the list of output or input events.

Defining Temporary Data

Stateflow allows stateless charts and graphical functions to define temporary data that persists only as long as the chart or graphical function is active. Only the parent chart or graphical function can access its temporary data. Defining a loop counter to have **Temporary** scope is a good idea since it is used only as a counter and does not need to persist.

Exporting Data

Stateflow can export definitions of Stateflow machine data to external code that embeds the machine. Exporting data enables external code, as well as the machine, to access the data. To export a data item, first add it to the data dictionary as the child of the Stateflow machine in which it is defined. Then set its **Scope** property to `Exported` and its other properties (for example, `Name` and `Type`) to appropriate values.

Note Only a Stateflow machine (see "machine" on page A-8) can parent external data. This means that you must use the Explorer to add external data to the data dictionary. It also means that external data are visible everywhere in the machine.

The Stateflow code generator generates a C declaration for each exported data item of the form

```
type ext_data;
```

where `type` is the C type of the exported item (for example, `int16`, `double`, and so on) and `data` is the item's Stateflow name. For example, suppose that your Stateflow machine defines an exported `integer` item named `counter`. The Stateflow code generator exports the item as the C declaration

```
int ext_counter;
```

The code generator includes declarations for exported data in the generated target's global header file, thereby making the declarations visible to external code compiled into or linked to the target.

Importing Data

A Stateflow machine can import definitions of data defined by external code that embeds the Stateflow machine. Importing externally defined data enables the machine to access data defined by the system in which it is embedded. To import an externally defined data item into a Stateflow machine, add a default item to the data dictionary as a child of the machine. Then set the new item's **Scope** property to `Imported`, its **Name** property to the name used by the machine's actions to reference the item, and its other properties (for example, **Type**, **Initial Value**, and so on) to appropriate values.

The Stateflow code generator assumes that external code provides a prototype for each imported item of the form

```
type ext_data;
```

where `type` is the C data type corresponding to the Stateflow data type of the imported item (for example, `int` for `Integer`, `double` for `Double`, and so on) and `data` is the item's Stateflow name. For example, suppose that your Stateflow machine defines an imported `integer` item named `counter`. The Stateflow code generator expects the item to be defined in the external C code as

```
int ext_counter;
```

7

Action Language

Stateflow attaches actions to a state or transition through its label. The following sections describe allowable contents for Stateflow action language:

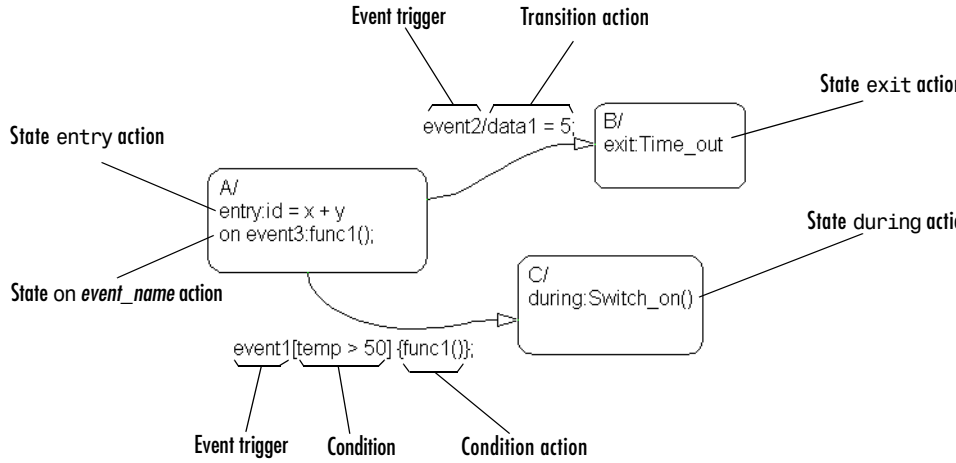
Types of Action Language (p. 7-3)	Gives a quick example of each type of action language available to Stateflow.
Keywords in Action Language (p. 7-4)	Describes the use of keywords to identify different types of actions.
Operations (p. 7-7)	Describes the available data operations in Stateflow action language.
Fixed-Point Data (p. 7-14)	Describes the fixed-point data types you can define for performing limited floating-point operations with integers on target platforms.
Calling C Functions (p. 7-42)	Describes the C functions that you can call directly in Stateflow action language.
MATLAB Functions and Data (p. 7-46)	Tells you how you can call MATLAB functions and access MATLAB workspace variables in action language, using the `ml` namespace operator or the `ml` function.
Data and Event Arguments (p. 7-58)	Tells you how to reference data defined at different levels of containment in a Stateflow chart when you use them as arguments for functions that you call in action language.
Arrays in Action Language (p. 7-60)	Describes how to use Stateflow data arrays in action language.
Event Broadcasting (p. 7-62)	Describes event broadcasting and directed event broadcasting in action language.
Condition Statements (p. 7-67)	Tells you how to write condition statements for taking a transition or executing an action in Stateflow action language.

Using Temporal Logic (p. 7-68)	You can test the occurrence of a specified multiple of events. Learn how to use temporal logic in Stateflow action language.
Special Symbols (p. 7-76)	Learn the special symbols Stateflow uses to provide the user with special features in action language notation.

Types of Action Language

Transitions can have event triggers, conditions, condition actions, and transition actions. States can have `entry`, `during`, `exit`, and `on event_name` actions. The action language for states and transitions represents these different types of actions using label notation.

The following diagram shows the action categories defined in the labels of states and transitions:



This chapter is a reference for the allowable contents of action language for states and transitions in Stateflow.

To learn about the notation and how to edit action language, see the following topics:

- "State Label Notation" on page 3-10
- "Transition Label Notation" on page 3-15
- "Labeling States" on page 5-23
- "Labeling Transitions" on page 5-30

To learn about the semantics (behavior) of action language, start with "Action Language" on page 3-42 and continue with "Stateflow Semantics" on page 4-1.

Keywords in Action Language

Stateflow action language (particularly for states) includes action language keywords that identify the different types of actions. For an understanding of each keyword, see the following topics:

- "State Action Type Keywords" on page 7-4 — Defines the leading keywords that specify action language categories in states.
- "Implicit Event Keywords" on page 7-5 — Defines the event keywords for events that occur when states are entered or exited or when a data value changes.
- "Special Function Keywords" on page 7-6 — Defines the keywords used to call special functions in action language.

Note Use of these keywords in any way other than their intended meaning within the rules of the notation will cause unpredictable results.

State Action Type Keywords

Each of the following keywords defines a category of action language notation for states. The short form of each keyword uses the characters displayed in bold type.

State Action Language Keyword	Meaning
du`ring`	Actions that follow are executed as part of a state's `during` action.
en`try`	Actions that follow are executed as part of a state's `entry` action.

State Action Language Keyword	Meaning
exit	Actions that follow are executed as part of a state's `exit` action.
on *event_name*	Actions that follow are executed when the `event_name` specified as an argument to the `on` keyword is broadcast.

Implicit Event Keywords

Each of the following keywords generates implicit events in the action language notation for states and transitions (see "Implicit Events" on page 6-12). The short form of each keyword uses the characters displayed in bold type.

State Action Language Keyword	Meaning
chan**g**e(data_name)	Specifies and implicitly generates a local event when the value of `data_name` changes.
enter(state_name)	Specifies and implicitly generates a local event when the specified `state_name` is entered.
exit(state_name)	Specifies and implicitly generates a local event when the specified `state_name` is exited.
tick	Same as `wakeup` keyword.
wakeup	Specifies and implicitly generates a local event when the chart of the action being evaluated awakens.

Special Function Keywords

Each of the keywords listed below call special functions in the action language notation for states and transitions.

Special Function Keyword	Meaning
`in(state_name)`	A condition function that is evaluated as true when the `state_name` specified as the argument is active.
`ml(evalString,arg1,arg2,...)`	Action specifies a call using `ml` function notation. See “ml Function” on page 7-47.
`ml.MATLAB_workspace_data`	Action specifies a call using the `ml` name space notation. See “ml Namespace Operator” on page 7-46.
`send(event_name,state_name)`	Send the `event_name` specified to the `state_name` specified (directed event broadcasting). See “Directed Event Broadcasting” on page 7-64.

Operations

Stateflow maintains a set of allowable operations between Stateflow data in action language. The following sections categorize the operations you can use in Stateflow action language:

- "Binary and Bitwise Operations" on page 7-7 — Lists and describes the supported action language operations that require two operands.
- "Unary Operations" on page 7-10 — Lists and describes the supported action language operations that require one operand.
- "Unary Actions" on page 7-11 — Lists and describes the supported action language operations that require one operand and an operator to the right.
- "Assignment Operations" on page 7-11 — Lists and describes the supported action language operations that assign the results of an operation to an operand.
- "Pointer and Address Operations" on page 7-12 — Lists and describes the supported action language operations that point to the location of data.
- "Typecast Operations" on page 7-13 — Lists and describes the supported action language operations that change the data type of an operand.

Binary and Bitwise Operations

The table that follows summarizes the interpretation of all binary operators in Stateflow action language. Table order gives relative operator precedence; highest precedence (10) is at the top of the table. Binary operators are evaluated left to right (left associative).

If you select the **Enable C-like bit operations** check box in the properties dialog for the chart (see the section "Specifying Chart Properties" on page 5-78), some of the binary operators are interpreted as bitwise operators. See individual operators in the table that follows for specific binary or bit operator interpretations.

Example	Precedence	Description
`a * b`	10	Multiplication
`a / b`	10	Division

Example	Precedence	Description
`a %% b`	10	Modulus
`a + b`	9	Addition
`a - b`	9	Subtraction
`a >> b`	8	Shift operand `a` right by `b` bits. Noninteger operands for this operator are first cast to integers before the bits are shifted.
`a << b`	8	Shift operand `a` left by `b` bits. Noninteger operands for this operator are first cast to integers before the bits are shifted.
`a > b`	7	Comparison of the first operand greater than the second operand
`a < b`	7	Comparison of the first operand less than the second operand
`a >= b`	7	Comparison of the first operand greater than or equal to the second operand
`a <= b`	7	Comparison of the first operand less than or equal to the second operand
`a == b`	6	Comparison of equality of two operands
`a ~= b`	6	Comparison of inequality of two operands
`a != b`	6	Comparison of inequality of two operands
`a <> b`	6	Comparison of inequality of two operands

Example	Precedence	Description
`a & b`	5	One of the following: • Bitwise AND of two operands Enabled when **Enable C-like bit operations** is selected in chart properties dialog. See "Specifying Chart Properties" on page 5-78. • Logical AND of two operands Enabled when **Enable C-like bit operations** is cleared in chart properties dialog.
`a ^ b`	4	One of the following: • Bitwise XOR of two operands Enabled when **Enable C-like bit operations** is selected in chart properties dialog. See "Specifying Chart Properties" on page 5-78. • Operand `a` raised to power `b` Enabled when **Enable C-like bit operations** is cleared in chart properties dialog. Use parentheses around power expressions with the `^` operator when used in conjunction with other arithmetic operators to avoid problems with operator precedence. For example, the action `z=x^2+y^2` should be rewritten as `z=(x^2)+(y^2)`.

Example	Precedence	Description
`a | b`	3	One of the following: • Bitwise OR of two operands Enabled when **Enable C-like bit operations** is selected in chart properties dialog. See "Specifying Chart Properties" on page 5-78. • Logical OR of two operands Enabled when **Enable C-like bit operations** is cleared in chart properties dialog.
`a && b`	2	Logical AND of two operands
`a || b`	1	Logical OR of two operands

Unary Operations

The following unary operators are supported in Stateflow action language. Unary operators have higher precedence than binary operators and are evaluated right to left (right associative).

Example	Description
`~a`	Logical NOT of `a` Complement of `a` (if `bitops` is enabled)
`!a`	Logical NOT of `a`
`-a`	Negative of `a`

Unary Actions

The following unary actions are supported in Stateflow action language.

Example	Description
`a++`	Increment `a`
`a--`	Decrement `a`

Assignment Operations

The following assignment operations are supported in Stateflow action language.

Example	Description
`a = expression`	Simple assignment
`a := expression`	Used primarily with fixed-point numbers. See "Assignment Operator :=" on page 7-32 for a detailed description.
`a += expression`	Equivalent to `a = a + expression`
`a -= expression`	Equivalent to `a = a - expression`
`a *= expression`	Equivalent to `a = a * expression`
`a /= expression`	Equivalent to `a = a / expression`

The following assignment operations are supported in Stateflow action language when **Enable C-like bit operations** is selected in the properties dialog for the chart. See "Specifying Chart Properties" on page 5-78.

Example	Description
`a |= expression`	Equivalent to `a = a | expression` (bit operation). See operation `a | b` in "Binary and Bitwise Operations" on page 7-7.
`a &= expression`	Equivalent to `a = a & expression` (bit operation). See operation `a & b` in "Binary and Bitwise Operations" on page 7-7.
`a ^= expression`	Equivalent to `a = a ^ expression` (bit operation). See operation `a ^ b` in "Binary and Bitwise Operations" on page 7-7.

Pointer and Address Operations

The address operator is available for use with both custom code variables and Stateflow variables. The pointer operator is available for use with custom code variables only.

Note The action language parser uses a relaxed set of restrictions. As a result, many syntax errors are not trapped until compilation.

The following examples show syntax that is valid for use with *custom code* variables only.

```
varStruct.field = <expression>;
(*varPtr) = <expression>;
varPtr->field = <expression>;
myVar = varPtr->field;
varPtrArray[index]->field = <expression>;
varPtrArray[expression]->field = <expression>;
myVar = varPtrArray[expression]->field;
```

The following examples show syntax that is valid for use with both custom code variables and Stateflow variables.

```
varPtr = &var;
ptr = &varArray[<expression>];
*(&var) = <expression>;
function(&varA, &varB, &varC);
function(&sf.varArray[<expr>]);
```

Typecast Operations

A *typecast operator* converts a value to a specified data type. Stateflow typecast operators have the same notation as MATLAB typecast operators,

```
op(v)
```

where `op` is the typecast operator (e.g, `int8`, `int16`, `int32`, `single`, `double`) and `v` is the value to be converted.

Normally you do not need to use typecast operators in actions. This is because Stateflow checks whether the types involved in a variable assignment differ and, if so, inserts a typecast operator in the generated code. (Stateflow uses the typecast operator of the language in which the target is generated, typically C.) However, if external code defines either or both types, Stateflow cannot determine which typecast, if any, is required. If a type conversion is necessary, you must use a Stateflow action language typecast operator to tell Stateflow which target language typecast operator to generate.

For example, suppose `varA` is a data dictionary value of type `double` and y is an external variable of type 32 bit integer. The following notation

```
y = int32(varA)
```

tells Stateflow to generate a typecast operator that converts the value of `varA` to a 32-bit integer before the value is assigned to `y`.

Fixed-Point Data

Fixed-point numbers use integers and integer arithmetic to approximate real numbers. They are an efficient means for performing computations involving real numbers without requiring floating-point support in underlying system hardware.

The following topics describe fixed-point data in Stateflow. Be sure to read "Tips and Tricks for Using Fixed-Point Data in Stateflow" on page 7-20 when you are ready to begin using it.

- "Fixed-Point Arithmetic" on page 7-15 — Provides a general discussion of the underlying arithmetic of fixed-point numbers.
- "How Stateflow Implements Fixed-Point Data" on page 7-17 — Describes the parameters that Stateflow uses to define fixed-point data.
- "Specifying Fixed-Point Data in Stateflow" on page 7-19 — Tells you where and how to specify fixed-point data in Stateflow.
- "Tips and Tricks for Using Fixed-Point Data in Stateflow" on page 7-20 — Gives you guidelines for using fixed-point data in Stateflow.
- "Offline and Online Conversions of Fixed-Point Data" on page 7-22 — Describes conversions of fixed-point numbers from one type to another that take place during code generation (offline) and execution (online).
- "Fixed-Point Context-Sensitive Constants" on page 7-23 — Tells you how to specify fixed-point constants that take their data type from the context in which they are used.
- "Supported Operations with Fixed-Point Operands" on page 7-24 — Provides a reference list of all supported operations with fixed-point numbers.
- "Promotion Rules for Fixed-Point Operations" on page 7-27 — Describes the data types automatically selected by Stateflow to receive the results of operations with fixed-point numbers.
- "Assignment (=, :=) and Cast Operations" on page 7-32 — Describes the data types that result from assignment operations with fixed-point numbers.
- "Overflow Detection for Fixed-Point Types" on page 7-37 — Tells you how to detect errors that result from exceeding the numeric capacity of fixed-point numbers.

- "Sharing Fixed-Point Data with Simulink" on page 7-37 — Tells you how to specify fixed-point data in Simulink that it shares with Stateflow as input from Simulink and output to Simulink data
- "Fixed-Point "Bang-Bang Control" Example" on page 7-38 — Examines an example using fixed-point data in Stateflow to perform limited floating-point operations on an 8 bit processor.

Fixed-Point Arithmetic

This section presents a high-level overview of fixed-point arithmetic with the following topics:

- "Fixed-Point Numbers" on page 7-15
- "Fixed-Point Operations" on page 7-16

Use this section to understand fixed-point numbers before you attempt to understand how Stateflow implements these numbers in "How Stateflow Implements Fixed-Point Data" on page 7-17.

For more discussion on the theory of fixed-point numbers, see "Fixed-Point Numbers" in the Fixed-Point Blockset documentation.

Fixed-Point Numbers

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V \approx \tilde{V} = SQ + B$$

where

- V is a precise real-world value that you want to approximate with a fixed-point number.
- $\tilde{V}$ is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes $\tilde{V}$. It is referred to as the *quantized integer*.

  Q is the actual stored integer value used in representing the fixed-point number; that is, if a fixed-point number changes, its quantized integer, Q, changes — S and B remain unchanged.
- S is a coefficient of Q referred to as the *slope*.

- B is an additive correction referred to as the *bias*.

Fixed-point numbers encode real quantities (for example, 15.375) using the stored integer Q. You set Q's value by solving the preceding equation $V = SQ + B$ for Q and rounding the result to an integer value as follows:

$$Q = round((V - B)/S)$$

For example, suppose you want to represent the number 15.375 in a fixed-point type with the slope $S = 0.5$ and the bias $B = 0.1$. This means that

$$Q = round((15.375 - 0.1)/0.5) = 30$$

However, because Q is rounded to an integer, you have lost some precision in representing the number 15.375. If you calculate the number that Q actually represents, you now get a slightly different answer.

$$V \approx \tilde{V} = SQ + B = 0.5 \cdot 30 + 0.1 = 15.1$$

So using fixed-point numbers to represent real numbers with integers involves the loss of some precision. However, if you choose S and B correctly, you can minimize this loss to acceptable levels.

Fixed-Point Operations

Now that you can express fixed-point numbers as $\tilde{V} = SQ + B$, you can define operations between two fixed-point numbers.

The general equation for an operation between fixed-point operands is as follows:

```
c = a <op> b
```

where `a`, `b`, and `c` are all fixed-point numbers, and `<op>` refers to one of the binary operations: addition, subtraction, multiplication, or division.

The general form for a fixed-point number `x` is $S_x Q_x + B_x$ (see "Fixed-Point Numbers" on page 7-15). Substituting this form for the result and operands in the preceding equation yields the following:

$$(S_c Q_c + B_c) = (S_a Q_a + B_a) \texttt{ < op > } (S_b Q_b + B_b)$$

The values for S_c and B_c are usually chosen by Stateflow for each operation (see "Promotion Rules for Fixed-Point Operations" on page 7-27) and are based

on the values for S_a, S_b, B_a, and B_b, which are entered for each fixed-point data (see "Specifying Fixed-Point Data in Stateflow" on page 7-19).

Note Stateflow also offers a more precise means for choosing the values for S_c and B_c when you use the `:=` assignment operator (that is, `c := a <op> b`). See "Assignment Operations" on page 7-11 for more detail.

Using the values for S_a, S_b, S_c, B_a, B_b, and B_c, you can solve the preceding equation for Q_c for each binary operation as follows:

- The operation `c=a+b` implies that
  $Q_c = ((S_a/S_c)Q_a + (S_b/S_c)Q_b + (B_a + B_b - B_c)/S_c)$
- The operation `c=a-b` implies that
  $Q_c = ((S_a/S_c)Q_a - (S_b/S_c)Q_b - (B_a - B_b - B_c)/S_c)$
- The operation `c=a*b` implies that
  $Q_c = ((S_aS_b/S_c)Q_aQ_b + (B_aS_b/S_c)Q_a + (B_bS_a/S_c)Q_a + (B_aB_b - B_c)/S_c)$
- The operation `c=a/b` implies that
  $Q_c = ((S_aQ_a + B_a)/(S_c(S_bQ_b + B_b)) - (B_c/S_c))$

The fixed-point approximations of the real number result of the operation `c = a <op> b` are given by the preceding solutions for the value Q_c. In this way, all fixed-point operations are performed using only the stored integer Q for each fixed-point number and integer operation.

How Stateflow Implements Fixed-Point Data

The preceding example in "Fixed-Point Arithmetic" on page 7-15 does not answer the question of how the values for the slope, S, the quantized integer, Q, and the bias, B, are implemented in Stateflow as integers. These values are implemented through the following:

- Stateflow defines a fixed-point data's type from values that you specify.

  You specify values for S, B, and the base integer type for Q. The available base types for Q are the unsigned integer types `uint8`, `uint16`, and `uint32`, and the signed integer types `int8`, `int16`, and `int32`. For specific

instructions on how to enter fixed-point data, see "Specifying Fixed-Point Data in Stateflow" on page 7-19.

Notice that if a fixed-point number has a slope $S = 1$ and a bias $B = 0$, it is equivalent to its quantized integer Q, and behaves exactly as its base integer type.

- Stateflow implements an integer variable for the Q value of each fixed-point data in generated code.

  This is the only part of a fixed-point number that varies in value. The quantities S and B are constant and appear only as literal numbers or expressions in generated code.

- The slope, S, is factored into an integer power of two, E, and a coefficient, F, such that $S = F*2^E$ and $1 \le F < 2$.

  The powers of 2 are implemented as bit shifts, which are more efficient than multiply instructions. Setting $F = 1$ avoids the computationally expensive multiply instructions for values of $F > 1$. This is referred to as *binary-point-only* scaling, which is implemented with bit shifts only, and is highly recommended.

- Operations for fixed-point types are implemented with solutions for the quantized integer as described in "Fixed-Point Operations" on page 7-16.

  To generate efficient code, the fixed-point promotion rules choose values for S_c and B_c that conveniently cancel out difficult terms in the solutions. See "Addition (+) and Subtraction (-)" on page 7-29 and "Multiplication (*) and Division (/)" on page 7-30.

  Stateflow provides a special assignment operator (`:=`) and context-sensitive constants to help you maintain as much precision as possible in your fixed-point operations. See "Assignment (=, :=) and Cast Operations" on page 7-32 and "Fixed-Point Context-Sensitive Constants" on page 7-23.

- Any remaining numbers, such as the fractional slope, F, that cannot be expressed as a pure integer or a power of 2, are converted into fixed-point numbers.

  These remaining numbers can be computationally expensive in multiplication and division operations. That is why the practice of using binary-point-only scaling in which $F = 1$ and $B = 0$ is recommended.

- During simulation, Stateflow detects when the result of a fixed-point operation *overflows* the capacity of its fixed-point type. See "Overflow Detection for Fixed-Point Types" on page 7-37.

Specifying Fixed-Point Data in Stateflow

You can specify fixed-point data in Stateflow as follows:

1. Add data to Stateflow as described in "Adding Data to the Data Dictionary" on page 6-14.

2. Set the properties for the data in the data properties dialog as described in "Setting Data Properties" on page 6-16. For fixed-point data, set the following fields:

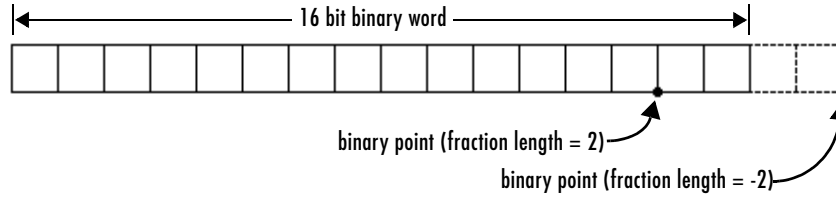
   - In the **Type** field, select **fixpt**.

     This enables the Fixed-Point field **Stored Integer** and optional fields **Fraction length** and **Scaling** directly below the **Type** field.

   - In the **Stored Integer** field, select the desired integer type to hold Q.

     Choose between the unsigned integer types `uint8`, `uint16`, and `uint32`, and the signed integer types `int8`, `int16`, `int32`.

   - If you want binary-point-only scaling for a fixed-point data (see note below), select the optional **Fraction length** field. You specify the number of bits to the right of the binary point in this fixed-point data by entering an integer value in the field to the right (noninteger values that you enter for **Fraction length** are flagged immediately with red text). A positive value moves the binary point left of the rightmost bit (least significant bit) by that amount. A negative value moves the binary point further right of the rightmost bit by that amount.

   - If you want to enter separate slope and bias values (see note below), select the **Scaling** option. In the field to the right, enter a two-element array,

that is, `[s b]`, in which the first element, `s`, is the slope, and the second element, `b`, is the bias. The slope must be greater than zero (values that you enter for **Fraction length** that are less than 0 are flagged immediately with red text).

For example, to specify the fixed-point data 2.4*Q + 3.4, enter the slope and bias as `[2.4 3.4]`in the **Scale** field. You can also enter the slope or bias as an arithmetic expression such as `[1.2*-2^1 3.4]`.

Note It is recommended that you use binary-point-only scaling whenever possible to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data using binary-point-only scaling are performed with simple bit shifts and eliminate the expensive code implementations required for separate slope and bias values.

You can also specify a fixed-point constant indirectly in action language by using a fixed-point context-sensitive constant. See "Fixed-Point Context-Sensitive Constants" on page 7-23.

Tips and Tricks for Using Fixed-Point Data in Stateflow

Once you specify fixed-point data (see "Specifying Fixed-Point Data in Stateflow" on page 7-19), you can use it just as you would any data in Stateflow. However, because of the limitations of fixed-point numbers, it is a good idea to follow these guidelines when using them:

1. Develop and test your application using double- or single-precision floating-point numbers.

   Using double- or single-precision floating-point numbers does not limit the range or precision of your computations. You need this while you are building your application.

2. Once your application works well, start substituting fixed-point data for double-precision data during the simulation phase, as follows:

a Set the integer word size for the simulation environment to the integer size of the intended target environment.

Stateflow uses this integer size in generated code to select result types for your fixed-point operations. See "Setting the Integer Word Size for a Target" on page 7-28.

b Add the suffix 'C' to literal numeric constants.

This suffix casts a literal numeric constant in the type of its context. For example, if `x` is fixed-point data, the expression `y = x/3.2C` first converts the numerical constant `3.2` to the fixed-point type of `x` and then performs the division with a fixed-point result. See "Fixed-Point Context-Sensitive Constants" on page 7-23 for more information.

Note If you do not use context-sensitive constants with fixed-point types, noninteger numeric constants (for example, constants that have a decimal point) can force fixed-point operations to produce floating-point results.

---

3 When you simulate, use overflow detection.

See "Overflow Detection for Fixed-Point Types" on page 7-37 for instructions on how to set overflow detection in simulation.

4 If you encounter overflow errors in fixed-point data, you can do one of the following to add range to your data.

- Increase the number of bits in the overflowing fixed-point data.

  For example, change the base type for Q from `int16` to `int32`.

- Increase the range of your fixed-point data by increasing the power of 2 value, E.

  For example, you might want to increase E from -2 to -1. This decreases the available precision in your fixed-point data.

5 If you encounter problems with model behavior stemming from inadequate precision in your fixed-point data, you can do one of the following to add precision to your data:

- Increase the precision of your fixed-point data by decreasing the value of the power of 2 binary point E.

  For example, you might want to decrease E from -2 to -3. This decreases the available range in your fixed-point data.
- If you decrease the value of E, you might also want to increase the number of bits in the base data type for Q to prevent overflow.

  For example, change the base type for Q from `int16` to `int32`.

6 If you cannot avoid overflow for lack of precision, consider using the `:=` assignment operator in place of the = operator for assigning the results of multiplication and division operations.

  You can use the `:=` operator to increase the range and precision of the result of fixed-point multiplication and division operations at the possible expense of computational efficiency. See "Assignment Operator :=" on page 7-32.

Offline and Online Conversions of Fixed-Point Data

Stateflow uses two different techniques for computing the quantized integer, Q, when converting real numbers into fixed-point quantities: offline and online.

Offline Conversion for Accuracy

Offline conversions are performed during code generation, and are designed to maximize accuracy. They round the resulting quantized integer to its nearest integer value. If the conversion overflows, the result saturates the value for Q.

Offline conversions are performed for the following operations:

- Initialization for data (both variables and constants) in the data dictionary
- Initialization of constants or variables from the MATLAB workspace

See "Offline and Online Conversion Examples" on page 7-23.

Online Conversion for Efficiency

Online conversions are performed during execution of the application and are designed to maximize computational efficiency. They are faster and more efficient than offline conversions, but less precise. Instead of rounding Q to its nearest integer, online conversions round to the floor (with the possible

exception of division, which might round to 0, depending on the C compiler you have). If the conversion overflows the type converted to, the result is undefined.

See "Offline and Online Conversion Examples" on page 7-23.

Offline and Online Conversion Examples

The following are examples of offline and online conversion of the fixed-point type defined by a 16 bit word size, a slope (S) equal to 2^{-4}, and a bias (B) equal to 0:

		Offline Conversion		**Online Conversion**	
V	V/S	Q	$\tilde{V}$	Q	$\tilde{V}$
3.45	55.2	55	3.4375	55	3.4375
1.0375	16.6	17	1.0625	16	1
2.06	32.96	33	2.0625	32	2

In the preceding example,

- V is the real-world value approximated by a fixed-point value.
- V/S is the floating-point computation for the quantized integer Q.
- Q is the rounded value of V/S.
- $\tilde{V}$ is the approximate real-world value resulting from Q for each conversion.

Fixed-Point Context-Sensitive Constants

You can conveniently use fixed-point constants without using the data properties dialog or Stateflow Explorer, by using context-sensitive constants. Context-sensitive constants are constants that infer their types from the context in which they occur. They are written like ordinary constants, but have the suffix `C` or `c`. For example, the numbers 4.3C and 123.4c are valid fixed-point context-sensitive constants you can use in action language operations.

Note Both operands of a binary operation cannot be context-sensitive constants.

While fixed-point context-sensitive constants can be used in context with any types (for example, `int32` or `double`), the primary motivation for using them is with fixed-point numbers. The algorithm that computes the type to assign to a fixed-point context-sensitive constant depends on the operator, the types in the context, and the value of the constant. It provides a "natural" type, providing maximum accuracy without overflow.

Supported Operations with Fixed-Point Operands

Stateflow supports the following operations for fixed-point operands:

Binary Operations

Stateflow supports the following binary operations with the listed precedence:

Example	Precedence	Description
`a * b`	10	Multiplication
`a / b`	10	Division
`a + b`	9	Addition
`a - b`	9	Subtraction
`a > b`	7	Comparison, greater than
`a < b`	7	Comparison, less than
`a >= b`	7	Comparison, greater than or equal to
`a <= b`	7	Comparison, less than or equal to
`a == b`	6	Comparison, equality
`a ~= b`	6	Comparison, inequality
`a != b`	6	Comparison, inequality

Example	Precedence	Description
`a <> b`	6	Comparison, inequality
`a & b`	5	One of the following: • Bitwise AND Enabled when **Enable C-like bit operations** is selected in the chart properties dialog. See "Specifying Chart Properties" on page 5-78. Operands are cast to integers before the operation is performed. • Logical AND Enabled when **Enable C-like bit operations** is cleared in chart properties dialog.
`a | b`	3	One of the following: • Bitwise OR Enabled when **Enable C-like bit operations** is selected in chart properties dialog. See "Specifying Chart Properties" on page 5-78. Operands are cast to integers before the operation is performed. • Logical OR Enabled when **Enable C-like bit operations** is cleared in chart properties dialog.
`a && b`	2	Logical AND
`a || b`	1	Logical OR

Unary Operations and Actions

Stateflow supports the following unary operations and actions:

Example	Description
`~a`	Unary minus
`!a`	Logical not
`a++`	Increment
`a--`	Decrement

Assignment Operations

Stateflow supports the following assignment operations:

Example	Description
`a = expression`	Simple assignment
`a := expression`	See "Assignment Operator :=" on page 7-32.
`a += expression`	Equivalent to `a = a + expression`
`a -= expression`	Equivalent to `a = a - expression`
`a *= expression`	Equivalent to `a = a * expression`
`a /= expression`	Equivalent to `a = a / expression`
`a |= expression`	Equivalent to `a = a | expression` (bit operation). See operation `a | b` in "Binary Operations" on page 7-24.
`a &= expression`	Equivalent to `a = a & expression` (bit operation). See operation `a & b` in "Binary Operations" on page 7-24.

Promotion Rules for Fixed-Point Operations

Operations with at least one fixed-point operand require rules for selecting the type of the intermediate result for that operation. For example, in the action statement `c = a + b`, where `a` or `b` is a fixed-point number, an intermediate result type for `a + b` must first be chosen before the result is calculated and assigned to `c`.

The rules for selecting the numeric types used to hold the results of operations with a fixed-point number are referred to as the *fixed-point promotion rules*. The primary goal of these rules is to maintain good computational efficiency with reasonable usability.

Note You can use the `:=` assignment operator to override the fixed-point promotion rules in the interest of obtaining greater accuracy. However, in this case, greater accuracy might require more computational steps. See "Assignment Operator :=" on page 7-32.

The following topics describe the process of selecting an intermediate result type for all binary operations with at least one fixed-point operand:

Default Selection of the Number of Bits of the Result Type

A fixed-point number with $S = 1$ and $B = 0$ is treated as an integer. In operations with integers, the C language promotes any integer input with fewer bits than the type `int` to the type `int` and then performs the operation.

The type `int` is the *integer word size* for C on a given platform. Result word size is increased to the integer word size because processors can perform operations at this size efficiently.

Stateflow maintains consistency with the C language by using the following default rule to assign the number of bits for the result type of an operation with fixed-point numbers:

- When both operands are fixed-point numbers, the number of bits in the result type is the maximum number of bits in the input types or the number of bits in the integer word size for the target machine, whichever is larger.

Note The preceding rule is a default rule for selecting the bit size of the result for operations with fixed-point numbers. This rule is overruled for specific operations as described in the sections that follow.

Setting the Integer Word Size for a Target. The preceding default rule for selecting the bit size of the result for operations with fixed-point numbers relies on the definition of the integer word size for your target. You can set the integer word size for the targets that you build in Simulink with the following procedure:

1. Select **Simulink parameters** from the **Simulink** menu in your Simulink model window.
2. In the resulting **Simulation Parameters** dialog, select the **Advanced** tab.
3. At the bottom of the **Advanced** panel, select **Microprocessor** for the **Production hardware characteristics** field.
4. In the window pane at the bottom of the panel, select **Number of bits for C 'int'** to highlight it.
5. In the **Value** field to the right, select the target integer size in bits.
6. Select **OK** to accept the changes.

When you build any target after making this change, Stateflow uses this integer size in generated code to select result types for your fixed-point operations.

Note It is recommended that you set all the available sizes in the **Value** field because they affect code generation, although they do not affect the implementation of the fixed-point promotion rules in generated code.

Unary Promotions

Only the unary minus (`-`) operation requires a promotion of its result type. The word size of the result is given by the default procedure for selecting the bit size of the result type for an operation involving fixed-point data. See “Default

Selection of the Number of Bits of the Result Type" on page 7-27. The bias, B, of the result type is the negative of the bias of the operand.

Binary Operation Promotion for Integer Operand with Fixed-Point Operand

Integers as operands in binary operations with fixed-point numbers are treated as fixed-point numbers of the same word size with slope, S, equal to 1, and a bias, B, equal to 0. The operation now becomes a binary operation between two fixed-point operands. See "Binary Operation Promotion for Two Fixed-Point Operands" on page 7-29.

Binary Operation Promotion for Double Operand with Fixed-Point Operand

When one operand is of type `double` in a binary operation with a fixed-point type, the result type is `double`. In this case, the fixed-point operand is cast to type `double`, and the operation is performed.

Binary Operation Promotion for Single Operand with Fixed-Point Operand

When one operand is of type `single` in a binary operation with a fixed-point type, the result type is `single`. In this case, the fixed-point operand is cast to type `single`, and the operation is performed.

Binary Operation Promotion for Two Fixed-Point Operands

Operations with both operands of fixed-point type produce an intermediate result of fixed-point type. The resulting fixed-point type is chosen through the application of a set of operator-specific rules. The procedure for producing an intermediate result type from an operation with operands of different fixed-point types is summarized in the following topics:

- "Setting the Integer Word Size for a Target" on page 7-28
- "Addition (+) and Subtraction (-)" on page 7-29
- "Multiplication (*) and Division (/)" on page 7-30
- "Relational Operations (>, <, >=, <=, ==, -=, !=, <>)" on page 7-30
- "Logical Operations (&, |, &&, ||)" on page 7-31

Addition (+) and Subtraction (-). The output type for addition and subtraction is chosen so that the maximum positive range of either input can be represented

in the output while preserving maximum precision. The base word type of the output follows the rule in "Default Selection of the Number of Bits of the Result Type" on page 7-27. To simplify calculations and yield efficient code, the biases of the two inputs are added for an addition operation and subtracted for a subtraction operation.

Note Mixing signed and unsigned operands might yield unexpected results and is not recommended.

Multiplication (*) and Division (/). The output type for multiplication and division is chosen to yield the most efficient code implementation. Nonzero biases are not supported for multiplication and division by Stateflow (see note).

The slope for the result type of the product of the multiplication of two fixed-point numbers is the product of the slopes of the operands. Similarly, the slope of the result type of the quotient of the division of two fixed-point numbers is the quotient of the slopes. The base word type is chosen to conform to the rule in "Default Selection of the Number of Bits of the Result Type" on page 7-27.

Note Because nonzero biases are computationally very expensive, they are not supported for multiplication and division by Stateflow.

Relational Operations (>, <, >=, <=, ==, -=, !=, <>). Stateflow supports the following relational (comparison) operations on all fixed-point types: >, <, >=, <=, ==, -=, !=, <>. See "Supported Operations with Fixed-Point Operands" on page 7-24 for an example and description of these operations. Stateflow requires that both operands in a comparison have equal biases (see note).

Comparing fixed-point values of different types can yield unexpected results because Stateflow must convert each operand to a common type for comparison. Because of rounding or overflow errors during the conversion, values that do not appear equal might be equal and values that appear to be equal might not be equal.

Note To preserve precision and minimize unexpected results, Stateflow requires both operands in a comparison operation to have equal biases.

For example, compare the following two unsigned 8 bit fixed-point numbers, `a` and `b`, in an 8 bit target environment:

Fixed-Point Number a	Fixed-Point Number b
$S_a = 2^{-4}$	$S_b = 2^{-2}$
$B_a = 0$	$B_b = 0$
$V_a = 43.8125$	$V_b = 43.75$
$Q_a = 701$	$Q_b = 175$

By rule, the result type for comparison is 8 bit. Converting `b`, the least precise operand, to the type of a, the most precise operand, could result in overflow. Consequently, `a` is converted to the type of `b`. Because the bias values for both operands are 0, the conversion is made as follows:

$$S_b(newQ_a) = S_aQ_a$$

$$newQ_a = (S_a/S_b)Q_a = (2^{-4}/2^{-2})701 = 701/4 = 175$$

Although they represent different values, `a` and `b` are considered equal as fixed-point numbers.

Logical Operations (&, |, &&, ||). If `a` is a fixed-point number used in a logical operation, it is interpreted with the equivalent substitution `a != 0.0C` where `0.0C` is an expression for zero in the fixed-point type of `a` (see "Fixed-Point Context-Sensitive Constants" on page 7-23). For example, if `a` is a fixed-point number in the logical operation `a && b`, this operation is equivalent to the following:

```
(a != 0.0C) && b
```

The preceding operation is not a check to see whether the quantized integer for a, Q_a, is not 0. If the real-world value for a fixed-point number `a` is 0, this

implies that $V_a = S_a Q_a + B_a = 0.0$. Therefore, the expression `a != 0`, for fixed-point number `a`, is actually equivalent to the following expression:

$$Q_a \; ! = -B_a / S_a$$

For example, if a fixed-point number, `a`, has a slope of 2^{-2}, and a bias of 5, the test `a != 0` is equivalent to the test `if` $Q_a \; ! = -20$.

Assignment (=, :=) and Cast Operations

Stateflow supports the assignment operations `LHS = RHS` and `LHS := RHS` between a left-hand side (`LHS`) and a right-hand side (`RHS`). These are described in the following topics:

- "Assignment Operator =" on page 7-32.
- "Assignment Operator :=" on page 7-32
- ":= Multiplication Example" on page 7-33
- ":= Division Example" on page 7-35
- ":= Assignment and Context-Sensitive Constants" on page 7-36

Assignment Operator =

An assignment statement of the type LHS = RHS is equivalent to casting the right-hand side to the type of the left-hand side. Stateflow supports any assignment between fixed-point types and therefore, implicitly, any cast.

A cast converts the stored integer Q from its original fixed-point type while preserving its value as accurately as possible using the online conversions (see "Offline and Online Conversions of Fixed-Point Data" on page 7-22). Assignments are most efficient when both types have the same bias, and slopes that are equal or both powers of 2.

Note No explicit cast operation is provided for fixed-point numbers.

Assignment Operator :=

Ordinarily, Stateflow uses the fixed-point promotion rules to choose the result type for an operation. Using the `:=` assignment operator overrides this behavior by using the type of the `LHS` as the result type of the `RHS` operation.

This type of assignment is particularly useful in retaining useful range and precision in the result of a multiplication or division that ordinary assignment might not retain. It is less useful with addition or subtraction but can avoid overflow or the loss of memory to store a result even in these cases.

Use of the `:=` assignment operator is governed by the following rules:

- The `RHS` can contain at most one binary operator.
- If the `RHS` contains anything other than a multiplication (*), division (/), addition (+), or subtraction (-) operation, or a constant, then the `:=` assignment behaves exactly like regular assignment (=).
- Constants on the `RHS` of an `LHS := RHS` assignment are converted to the type of the left-hand side using offline conversion (see "Offline and Online Conversions of Fixed-Point Data" on page 7-22). Ordinary assignment always casts the `RHS` using online conversions.

For examples contrasting the `LHS := RHS` and the `LHS = RHS` assignment operations, see the following:

- ":= Multiplication Example" on page 7-33
- ":= Division Example" on page 7-35

Caution Using the `:=` assignment operator to produce a more accurate result might generate code that is less efficient than the code generated using the normal fixed-point promotion rules.

:= Multiplication Example

The following example contrasts the `:=` and `=` assignment operators for multiplication. Here, the `:=` operator is used to avoid overflow in the results of the multiplication `c = a * b` in which `a` and `b` are of two fixed-point operands.

The operands and result for this operation are 16 bit unsigned integers with the following assignments:

Fixed-Point Number a	Fixed-Point Number b	Fixed-Point Number c
$S_a = 2^{-4}$	$S_b = 2^{-4}$	$S_c = 2^{-5}$
$B_a = 0$	$B_b = 0$	$B_c = 0$
$V_a = 20.1875$	$V_b = 15.3125$	$V_c = ?$
$Q_a = 323$	$Q_b = 245$	$Q_c = ?$

where S is the slope, B is the bias, V is the real-world value, and Q is the quantized integer.

`c = a*b.` In this case, first calculate an intermediate result for `a*b` in the fixed-point type given by the rules in the section "Fixed-Point Operations" on page 7-16, and then cast that result into the type for `c`.

The intermediate value is calculated as follows:

$$\begin{aligned} Q_{iv} &= Q_a Q_b \\ &= 323 \cdot 245 = 79135 \end{aligned}$$

Because the maximum value of a 16 bit unsigned integer is $2^{16} - 1 = 65535$, the preceding result overflows its word size. An operation that overflows its type produces an undefined result.

You can capture overflow errors like the preceding example during simulation with the Debugger window. See "Overflow Detection for Fixed-Point Types" on page 7-37.

`c := a*b.` In this case, calculate `a*b` directly in the type of `c`. Use the solution for Q_c given in "Fixed-Point Operations" on page 7-16 with the requirement of zero bias, which is as follows:

$$\begin{aligned} Q_c &= ((S_a S_b / S_c) Q_a Q_b) \\ &= (2^{-4} \cdot 2^{-4} / 2^{-5})(323 \cdot 245) \\ &= 79135/8 = 9892 \qquad \text{(rounded to floor)} \end{aligned}$$

No overflow occurs in this case, and the approximate real-world value is as follows:

$$\tilde{V}_c = S_c Q_c = 2^{-5} \cdot 9892 = 9892/32 = 309.125$$

This value is very close to the actual real-world result of 309.121.

:= Division Example

The following example contrasts the `:=` and `=` assignment operators for division. The `:=` operator is used to obtain more precise results for the division of two fixed-point operands, `b` and `c`, in the statement `c := a/b`.

This example uses the following fixed-point numbers, where S is the slope, B is the bias, V is the real-world value, and Q is the quantized integer:

Fixed-Point Number a	Fixed-Point Number b	Fixed-Point Number c
$S_a = 2^{-4}$	$S_b = 2^{-3}$	$S_c = 2^{-6}$
$B_a = 0$	$B_b = 0$	$B_c = 0$
$V_a = 2$	$V_b = 3$	$V_c = ?$
$Q_a = 32$	$Q_b = 24$	$Q_c = ?$

c = a/b. In this case, first calculate an intermediate result for `a/b` in the fixed-point type given by the rules in the section "Fixed-Point Operations" on page 7-16, and then cast that result into the type for `c`.

The intermediate value is calculated as follows:

$$Q_{iv} = Q_a / Q_b$$

$$= 32/24 = 1$$

The intermediate value is then cast to the result type for `c` as follows:

$$S_c Q_c = S_{iv} Q_{iv}$$

$$Qc = (S_{iv}/Sc) Q_{iv}$$

The slope of the intermediate value for a division operation is calculated as

$$S_{iv} = S_a / S_b = 2^{-4} / 2^{-3} = 2^{-1}$$

Substitution of this value into the preceding result yields the final result.

$$Q_c = 2^{-1} / 2^{-6} = 2^5 = 32$$

In this case, the approximate real-world value is $\tilde{V}c = 32/64 = 0.5$, which is not a very good approximation of the actual result, $2/3 = 0.667$.

c := a/b. In this case, calculate `a/b` directly in the type of `c`. Use the solution for Q_c given in "Fixed-Point Operations" on page 7-16 with the simplification of zero bias, which is as follows:

$$\begin{aligned} Qc &= (S_a Q_a)/(S_c(S_b Q_b)) \\ &= (S_a/(S_b \cdot S_c)) \cdot Q_a / Q_b \\ &= (2^{-4}/(2^{-3} \cdot 2^{-6})) \cdot 32/24 \\ &= 2^5 \cdot 32/24 = 42 \end{aligned}$$

In this case, the approximate real-world value $\tilde{V}_c = 42/64 = 0.6563$, a much better approximation to the precise result, $2/3 = 0.667$.

:= Assignment and Context-Sensitive Constants

In a `:=` assignment operation, the type of the left-hand side (`LHS`) determines part of the context used for inferring the type of a right-hand side (`RHS`) context-sensitive constant.

The following rules apply to `RHS` context-sensitive constants in assignments with the `:=` operator:

- If the `LHS` is a floating-point data (type `double` or `single`) , the `RHS` context-sensitive constant becomes a floating-point constant.
- For addition and subtraction, the type of the LHS determines the type of the context-sensitive constant on the RHS.
- For multiplication and division, the type of the context-sensitive constant is chosen independent of the LHS.

Overflow Detection for Fixed-Point Types

Overflow occurs when the magnitude of a result assigned to a data exceeds the numeric capacity of that data. You enable Stateflow to detect overflow of integer and fixed-point operations during simulation with the following steps:

1. For the simulation target of your model, open the **Code Generation Options** dialog.

   See "Accessing the Target Builder Dialog for a Target" on page 11-9 for instructions on how to access this dialog.

2. Select both the **Enable debugging/animation** and **Enable overflow detection (for debugging)** options.

   For descriptions of these options, see "Specifying Code Generation Options" on page 11-11.

3. Build the simulation target.

   See "Starting a Simulation Target Build" on page 11-25 for instructions.

4. Open the Debugger window and select the **Data Range** check box.

   See "Error Checking Options" on page 12-8 for a description of this option.

5. Begin simulation.

Simulation breaks execution when an overflow occurs.

Sharing Fixed-Point Data with Simulink

If you plan on sharing fixed-point data with Simulink, use one of the following methods:

- Define the data that you input from Simulink or output to Simulink identically in both Stateflow and Simulink.

  This means that the values that you enter for the **Stored Integer** and **Scaling** fields in the shared data's properties dialog in Stateflow (see "Specifying Fixed-Point Data in Stateflow" on page 7-19) must match similar fields that you enter for fixed-point data in Simulink. See "Fixed-Point

"Bang-Bang Control" Example" on page 7-38 for an example of this method of sharing input from Simulink data using a Gateway In block in Simulink.

For some Simulink blocks, you can specify the type of input or output data directly. For example, you can set fixed-point output data directly in the block parameters dialog of the Simulink Constant block when you select **Specify via dialog** for the **Output data type mode** field (under **Show additional parameters**).

- Define the data as **Input from Simulink** or **Output to Simulink** in the data's properties dialog in Stateflow and instruct the sending or receiving block in Simulink to inherit its type from Stateflow.

  Many blocks allow you to set their data types and scaling through inheritance from the driving block, or through backpropagation from the next block. This can be a good way to set the data type of a Simulink block to match the data type of the Stateflow port it connects to.

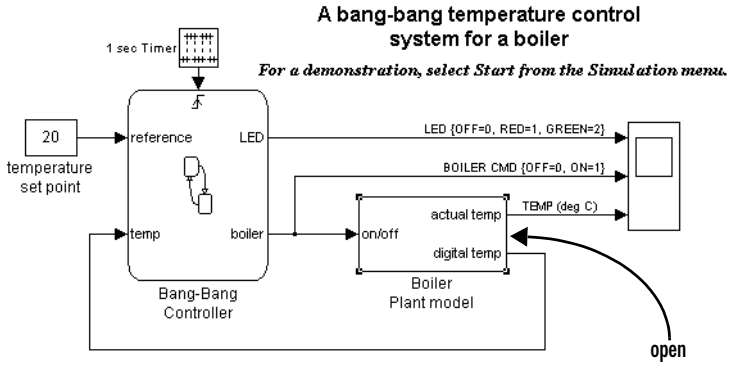
  For example, you can set the Simulink Constant block to inherit its type from the Stateflow **Input to Simulink** port that it supplies by selecting **Inherit via back propagation** for the **Output data type mode** field in its block parameters dialog (under **Show additional parameters**).

For a general discussion of data compatibility between Simulink blocks, see "Unified Simulink and Fixed-Point Blockset Blocks" and "Compatibility with Simulink Blocks" in the Fixed-Point Blockset documentation.

Fixed-Point "Bang-Bang Control" Example

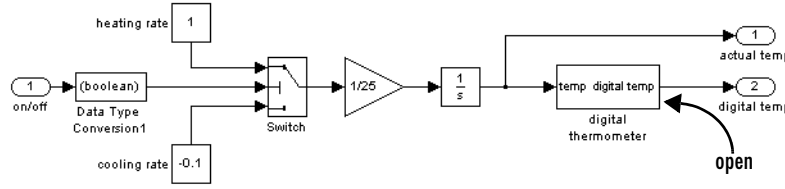
Stateflow includes demo models with applications of fixed-point data. For this example, load the `sf_boiler` demo model ("Bang-Bang control using Temporal Logic") into Simulink with the following steps:

1. In the MATLAB Launch Pad, expand the Simulink node.
2. Continue by expanding the Stateflow node.
3. Double-click the demos node under Stateflow.
4. In the resulting online Help screen, double-click the node marked "Bang-Bang control using temporal logic."



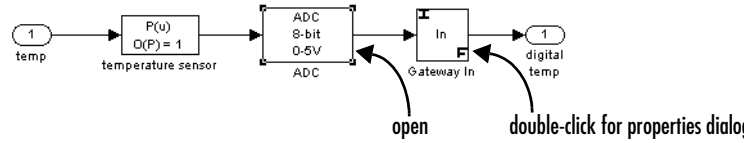
A Stateflow block performs almost all the logic of the bang-bang boiler model with the exception of the Boiler Plant model subsystem block.

5 Double-click the Boiler Plant model subsystem block.



The Boiler Plant model block simulates the temperature reaction of the boiler to periods of heating or cooling dictated by the Stateflow block. Depending on the Boolean value coming from the Controller, a temperature increment (+1 for heating, -0.1 for cooling) is added to the previous boiler temperature. The resulting boiler temperature is sent to the digital thermometer subsystem block.

6 Double-click the digital thermometer subsystem block.



The digital thermometer subsystem produces an 8 bit fixed-point representation of the input temperature with the blocks described in the sections that follow.

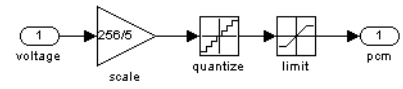
temperature sensor Block

The temperature sensor block converts input boiler temperature (T) to an intermediate analog voltage output T_{volts} with a first-order polynomial that results in the following output:

$$T_{volts} = 05*T + 75$$

ADC Block

Double-click the ADC block to reveal the following contents:



The ADC subsystem digitizes the analog voltage from the temperature sensor block by multiplying the analog voltage by $256/5$, rounding it to its integer floor, and limiting it to a maximum of 255 (the largest unsigned 8 bit integer value). Using the value for the output T_{volts} from the temperature sensor block, the new digital coded temperature output by the ADC block, $T_{digital}$, is given by the following equation:

$$T_{digital} = (265/5)*T_{volts} = (256*0.05/5)*T + (256/5)*0.75$$

Gateway In Block

An examination of the Block Parameters dialog for the Gateway In block shows that it informs the rest of the model that $T_{digital}$ is now a fixed-point number with a slope value of $5/256/0.05$ and an intercept value of $-0.75/0.05$. The Stateflow block Bang Bang Controller receives this output and interprets it as

a fixed-point number through the Stateflow data `temp`, which is scoped as **Input from Simulink** and set as an unsigned 8 bit fixed-point data with the same values for S and B set in the Gateway In block.

The values for S and B are determined from the general expression for a fixed-point number, which is as follows:

$$V = S*Q + B$$

Therefore,

$$Q = (V - B)/S = (1/S)*V + (-1/S)*B$$

Since $T_{digital}$ is now a fixed-point number, it is now the quantized integer Q of a fixed-point type. This means that $T_{digital} = Q$ of its fixed-point type and results in the following identity:

$$(1/S)*V + (-1/S)*B = (256*0.05/5)*T + (256/5)*0.75$$

Since T is the real-world value for the environment temperature, the above equation implies the following identifications:

$$V = T$$

and

$$1/S = (256*0.05)/5$$

$$S = 5/(256*0.05) = 0.390625$$

and

$$(-1/S)*B = (256/5)*0.75$$

$$B = -(256/5)*0.75*5/(256*0.05) = -0.75/0.05 = 15$$

By setting $T_{digital}$ to be a fixed-point data both as the output of the Gateway In block in Simulink and the input of the Stateflow Bang Bang Controller block, Stateflow interprets and processes this data automatically in an 8 bit environment with no need for any explicit conversions.

Calling C Functions

This section describes the C functions that you can call directly in Stateflow action language. See the following topics:

- "Calling C Library Functions" on page 7-42 — Tells you how to call C functions from the math library of your compiler or from custom libraries that you provide.
- "Calling min and max Functions" on page 7-42 — Describes the special min and max macros that you can call.
- "Calling User-Written C Code Functions" on page 7-43 — Tells you how to call C functions you provide in custom code for the target you build.

Calling C Library Functions

You can call the following small subset of the C Math Library functions:

`abs`	`acos`	`asin`	`atan`	`atan2`	`ceil`
`cos`	`cosh`	`exp`	`fabs`	`floor`	`fmod`
`labs`	`ldexp`	`log`	`log10`	`pow`	`rand`
`sin`	`sinh`	`sqrt`	`tan`	`tanh`	

You can call the above C Math Library functions without doing anything special as long as you are careful to call them with the right data types.

If you call other C library functions not specified above, be sure to include the appropriate `#include...` directive in the **Custom code included at the top of generated cod**e field of the **Target Options** dialog. See the section "Specifying Custom Code Options" on page 11-20.

Calling min and max Functions

Although `min` and `max` are not C library functions, Stateflow enables them by emitting the following macros automatically at the top of generated code.

```
#define min(x1,x2) ((x1) > (x2))? (x2):(x1)
#define max(x1,x2) ((x1) > (x2))? (x1):(x2)
```

To allow compatibility with user graphical functions named `min()` or `max()`, Stateflow generates code for them with a mangled name of the following form: `<prefix>_min`. However, if you export `min()` or `max()` graphical functions to other Stateflow charts in the Stateflow machine, the name of these functions can no longer be emitted with mangled names in generated code and conflict occurs. To avoid this conflict, rename the `min()` and `max()` graphical functions.

Calling User-Written C Code Functions

To install your own C code functions for use in Stateflow action language, do the following:

1. From the Tools menu, select the **Open** (**RTW** or **Simulation**) **Target** dialog.
2. When the **Open Target** dialog appears, select **Target Options**.
3. Enter the following:

- Include a header file containing the declarations of your C code functions in the **Custom code included at the top of generated cod**e field.
- Specify the source `file name` containing your C code functions in the **Custom source files** field.

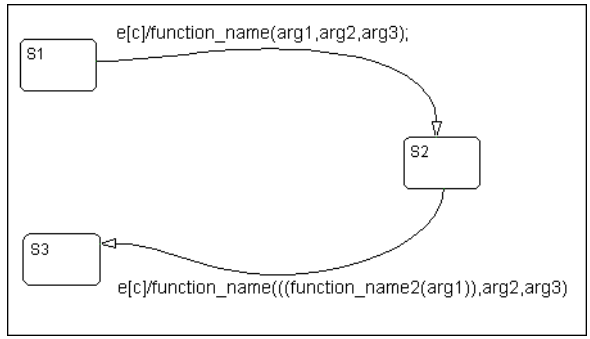
See “Specifying Custom Code Options” on page 11-20.

To use your own C code functions in Stateflow action language, follow these guidelines:

- Define a function by its name, any arguments in parentheses, and an optional semicolon.
- String parameters to user-written functions are passed between single quotation marks. For example, `func( string )`.
- An action can nest function calls.
- An action can invoke functions that return a scalar value (of type `double` in the case of MATLAB functions and of any type in the case of C user-written functions).

Function Call Transition Action Example

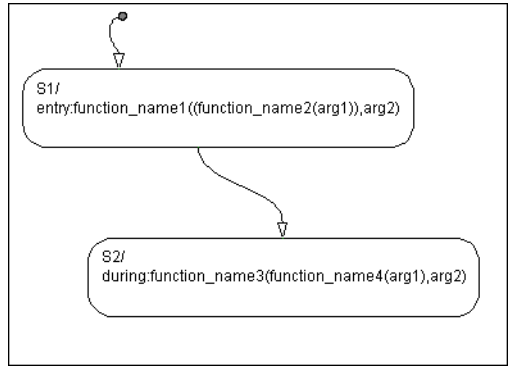
These are example formats of function calls using transition action notation.



If `S1` is active, event `e` occurs, `c` is true, and the transition destination is determined, then a function call is made to `function_name` with `arg1`, `arg2`, and `arg3`. The transition action in the transition from `S2` to `S3` shows a function call nested within another function call.

Function Call State Action Example

These are example formats of function calls using state action notation.



When the default transition into `S1` occurs, `S1` is marked active and then its `entry` action, a function call to `function_name1` with the specified arguments, is executed and completed. If `S2` is active and an event occurs, the `during` action, a function call to `function_name3` with the specified arguments, executes and completes.

Passing Arguments by Reference

A Stateflow action can pass arguments to a user-written function by reference rather than by value. In particular, an action can pass a pointer to a value rather than the value itself. For example, an action could contain the following call

```
f(&x);
```

where `f` is a custom-code C function that expects a pointer to `x` as an argument.

If `x` is the name of a data item defined in the Stateflow data dictionary, the following rules apply.

- Do not use pointers to pass data items input from Simulink.

  If you need to pass an input item by reference, for example, an array, assign the item to a local data item and pass the local item by reference.
- If `x` is a Simulink output data item having a data type other than `double`, the chart property **Use strong data typing with Simulink IO** must be on (see "Specifying Chart Properties" on page 5-78).
- If the data type of `x` is `boolean`, the coder option **Use bitsets to store state-configuration** must be turned off (see "Use bitsets for storing state configuration — Use bitsets for storing state configuration variables. This can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets." on page 11-13).
- If `x` is an array with its first index property set to 0 (see "Array" on page 6-21), then the function must be called as follows.

  ```
  f(&(x[0]));
  ```

  This passes a pointer to the first element of `x` to the function.
- If `x` is an array with its first index property set to a nonzero number (for example, 1), the function must be called in the following way:

  ```
  f(&(x[1]));
  ```

  This passes a pointer to the first element of `x` to the function.

MATLAB Functions and Data

You can call MATLAB functions and access MATLAB workspace variables in action language, using the `ml` namespace operator or the `ml` function. See the following sections:

- "ml Namespace Operator" on page 7-46 — Shows you how to use MATLAB workspace variables or call MATLAB functions through the `ml` namespace operator.
- "ml Function" on page 7-47 — Shows you how to use MATLAB functions through the `ml` function.
- "ml Expressions" on page 7-49 — Shows you how to mix ml namespace operator and `ml` function expressions along with Stateflow data in larger expressions.
- "ml Data Type" on page 7-51 — Shows you how to use the MATLAB data type to keep MATLAB data in Stateflow instead of in the MATLAB workspace.
- "Inferring Return Size for ml Expressions" on page 7-53 — Gives you the rules for providing enough information to infer the size of the values returned from the `ml` namespace operator and the ml function in larger action language expressions.

Caution Because MATLAB functions are not available in a target environment, do not use the `ml` namespace operator and the `ml` function if you plan to build an RTW target that includes code from Stateflow Coder.

---

ml Namespace Operator

The `ml` namespace operator uses standard dot (.) notation to reference MATLAB variables and functions in action language. For example, the statement `y = ml.x` returns the value of the MATLAB workspace variable `x` to the Stateflow data `y`. The statement `y = ml.matfunc(arg1, arg2)` passes the return value from the MATLAB function `matfunc` to Stateflow data `y`.

If the MATLAB function you call does not require arguments, you must still include the parentheses, as shown in the preceding examples. If the parentheses are omitted, Stateflow interprets the function name as a

workspace variable, which, when not found, generates a run-time error during simulation.

In the following examples, `x`, `y`, and `z` are workspace variables and `d1` and `d2` are Stateflow data:

- `a = ml.sin(ml.x)`

  In this example, the `sin` function of MATLAB evaluates the sine of `x`, which is then assigned to Stateflow data variable `a`. However, because `x` is a workspace variable, you must use the namespace operator to access it. Hence, `ml.x` is used instead of just `x`.

- `a = ml.sin(d1)`

  In this example, the `sin` function of MATLAB evaluates the sine of `d1`, which is assigned to Stateflow data variable `a`. Because `d1` is Stateflow data, you can access it directly.

- `ml.x = d1*d2/ml.y`

  The result of the expression is assigned to `x`. If `x` does not exist prior to simulation, it is automatically created in the MATLAB workspace.

- `ml.v[5][6][7] = ml.matfunc(ml.x[1][3], ml.y[3], d1, d2, 'string')`

  The workspace variables `x` and `y` are arrays. `x[1][3]` is the `(1,3)` element of the two-dimensional array variable `x`. `y[3]` is the third element of the one-dimensional array variable `y`. The last argument, `'string'`, is a literal string.

  The return from the call to `matfunc` is assigned to element `(5,6,7)` of the workspace array, `v`. If `v` does not exist prior to simulation, it is automatically created in the MATLAB workspace.

ml Function

You can use the `ml` function to specify calls to MATLAB functions through a string expression in the action language. The format for the `ml` function call uses standard function notation as follows:

```
ml(evalString, arg1,arg2,...);
```

`evalString` is a string expression that is evaluated in the MATLAB workspace. It contains a MATLAB command (or a set of commands, each separated by a semicolon) to execute along with format specifiers (`%g`, `%f`, `%d`, etc.) that provide

formatted substitution of the other arguments (`arg1`, `arg2`, etc.) into `evalString`.

The format specifiers used in `ml` functions are the same as those used in the C functions `printf` and `sprintf`. The `ml` function call is equivalent to calling the MATLAB `eval` function with the `ml` namespace operator if the arguments `arg1,arg2,...` are restricted to scalars or string literals in the following command:

```
ml.eval(ml.sprintf(evalString,arg1,arg2,...))
```

Stateflow assumes scalar return values from `ml` namespace operator and `ml` function calls when they are used as arguments in this context. See "Inferring Return Size for ml Expressions" on page 7-53.

In the following examples, `x` is a MATLAB workspace variable, and `d1` and `d2` are Stateflow data:

- `a = ml('sin(x)')`

  In this example, the `ml` function calls the `sin` function of MATLAB to evaluate the sine of `x` in the MATLAB workspace. The result is then assigned to Stateflow data variable `a`. Because `x` is a workspace variable, and `sin(x)` is evaluated in the MATLAB workspace, you enter it directly in the `evalString` argument (`'sin(x)'`).

- `a = ml('sin(%f)', d1)`

  In this example, the `sin` function of MATLAB evaluates the sine of `d1` in the MATLAB workspace and assigns the result to Stateflow data variable `a`. Because `d1` is Stateflow data, its value is inserted in the `evalString` argument (`'sin(%f)'`) using the format expression `%f`. This means that if `d1 = 1.5`, the expression evaluated in the MATLAB workspace is `sin(1.5)`.

- `a = ml('matfunc(%g, ''abcdefg'', x, %f)', d1, d2)`

  In this example, the string `'matfunc(%g, ''abcdefg'', x, %f)'` is the `evalString` shown in the preceding format statement. Stateflow data `d1` and d2 are inserted into that string with the format specifiers `%g` and `%f`, respectively. The string `''abcdefg''` is a string literal with two single pairs of quotation marks used to enclose it because it is part of the evaluation string, which is already enclosed in single quotation marks.

- `sfmat_44 = ml('rand(4)')`

  In this example, a square 4-by-4 matrix of random numbers between 0 and 1 is returned and assigned to the Stateflow data `sf_mat44`. Stateflow data `sf_mat44` must be defined in Stateflow as a 4-by-4 array before simulation. If its size is different, a size mismatch error is generated during run-time.

ml Expressions

You can mix `ml` namespace operator and `ml` function expressions along with Stateflow data in larger expressions. The following example squares the `sine` and `cosine` of an angle in workspace variable `X` and adds them:

```
ml.power(ml.sin(ml.X),2) + ml('power(cos(X),2)')
```

The first operand uses the `ml` namespace operator to call the `sin` function. Its argument is `ml.X`, since `X` is in the MATLAB workspace. The second operand uses the `ml` function. Because `X` is in the workspace, it is included in the `evalString` expression as `X`. The squaring of each operand is performed with the MATLAB power function, which takes two arguments: the value to square, and the power value, 2.

Expressions using the `ml` namespace operator and the `ml` function can be used as arguments for `ml` namespace operator and `ml` function expressions. The following example nests `ml` expressions at three different levels:

```
a = ml.power(ml.sin(ml.X + ml('cos(Y)')),2)
```

In composing your `ml` expressions, follow the levels of precedence set out in "Binary and Bitwise Operations" on page 7-7. To repeat a warning note in that section, be sure to use parentheses around power expressions with the `^` operator when you use them in conjunction with other arithmetic operators.

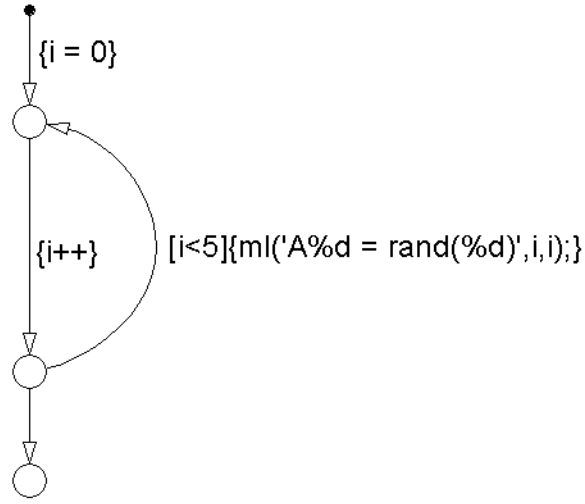
Stateflow checks expressions for data size mismatches in your action language during parsing of your charts and during run-time. Because the return values for `ml` expressions are not known till run-time, Stateflow must infer the size of their return values. See "Inferring Return Size for ml Expressions" on page 7-53.

Which ml Should I Use?

In most cases the notation of the `ml` namespace operator is more straightforward. However, using the `ml` function call does offer a few advantages:

- Use the `ml` function to dynamically construct workspace variables.

  The following example creates four new MATLAB matrices:



  This example demonstrates the use of a Stateflow `for` loop to create four new matrix workspace variables in MATLAB. The default transition initializes the Stateflow counter `i` to 0 while the transition segment between the top two junctions increments it by 1. If `i` is less than 5, the transition segment back to the top junction is taken and evaluates the `ml` function call `ml('A%d = rand(%d)',i,i)` for the current value of i. When i is greater than or equal to 5, the transition segment between the bottom two junctions is taken and execution stops.

  This results in the following MATLAB commands, which create a workspace scalar (A1) and three matrices (A2, A3, A4):

```
A1 = rand(1)
A2 = rand(2)
A3 = rand(3)
```

```
A4 = rand(4)
```

- Use the `ml` function with full MATLAB notation.

  You cannot use full MATLAB notation with the `ml` namespace operator, as demonstrated by the following example:

  ```
  ml.A = ml.magic(4)
  B = ml('A + A'''')
  ```

  This example sets the workspace variable `A` to a magic 4-by-4 matrix using the `ml` namespace operator. Stateflow data `B` is then set to the addition of `A` and its transpose matrix, `A'`, which produces a symmetric matrix. Because the `ml` namespace operator cannot evaluate the expression `A'`, the `ml` function is used instead. However, you can call the MATLAB function `transpose` with the `ml` namespace operator in the following equivalent expression:

  ```
  B = ml.A + ml.transpose(ml.A)
  ```

  As another example, you cannot use arguments with cell arrays or subscript expressions involving colons with the `ml` namespace operator. However, these can be included in an `ml` function call.

ml Data Type

Stateflow data of type `ml` is typed internally with the MATLAB type `mxArray`. This means that you can assign (store) any type of data available in Stateflow to a data of type `ml`. This includes any type of data defined in Stateflow or returned from MATLAB with the `ml` namespace operator or `ml` function.

The following features and limitations apply to Stateflow data of type `ml`.

- `ml` data can be initialized from the MATLAB workspace just like other data in Stateflow (see the **Initialize from** property in "Setting Data Properties" on page 6-16).
- Any numerical scalar or array of `ml` data in Stateflow can participate in any kind of unary operation and any kind of binary operation with any other data in Stateflow.

  If `ml` data participates in any numerical operation with other data, the size of the `ml` data must be inferred from the context in which it is used, just as return data from the `ml` namespace operator and `ml` function are. See "Inferring Return Size for ml Expressions" on page 7-53.

Note The preceding feature does not apply to `ml` data storing MATLAB 64 bit integers. `ml` data can store 64 bit MATLAB integers but cannot be used in Stateflow action language.

- `ml` data cannot be defined with the scope **Constant** or specified as an array.

  These options are disabled in the data properties dialog and Stateflow Explorer for Stateflow data of type `ml`.
- `ml` data can be used in building a simulation target (`sfun`) but not in building an RTW target (`rtw`).
- If data of type `ml` contains an array, the elements of the array can be accessed through indexing with the following rules:
  a. Only arrays with numerical elements can be indexed.
  b. Numerical arrays can be indexed according to their dimension only.

     This means that only one-dimensional arrays can be accessed by a single index value. You cannot access a multidimensional array with a single index value.
  c. The first index value for each dimension of an array is 1, and not 0, as in C-language arrays.

  In the examples that follow, `mldata` is a Stateflow data of type `ml`, `ws_num_array` is a 2-by-2 MATLAB workspace array with numerical values, and `ws_str_array` is a 2-by-2 MATLAB workspace array with string values.

```
mldata = ml.ws_num_array; /* OK */
n21 = mldata[2][1]; /* OK for numerical data of type ml */
n21 = mldata[3]; /* NOT OK for 2-by-2 array data */
mldata = ml.ws_str_array; /* OK */
s21 = mldata[2][1]; /* NOT OK for string data of type ml*/
```

- `ml` data cannot have a scope outside Stateflow; that is, it cannot be scoped as **Input to Simulink** or **Output to Simulink**.

Place Holder for Workspace Data

Both the `ml` namespace operator and the `ml` function can access data directly in the MATLAB workspace and return it to Stateflow. However, maintaining

data in the MATLAB workspace can present Stateflow users with conflicts with other data already resident in the workspace. Consequently, with the `ml` data type, you can maintain `ml` data in Stateflow and use it for MATLAB computations in Stateflow action language.

As an example, in the following Stateflow action language statements, `mldata1` and `mldata2` are Stateflow data of type `ml`:

```
mldata1 = ml.rand(3);
mldata2 = ml.transpose(mldata1);
```

In the first line of this example, `mldata1` receives the return value of the `rand` function in MATLAB, which, in this case, returns a 3-by-3 array of random numbers. Note that `mldata1` is not specified as an array or sized in any way. It can receive any MATLAB workspace data or the return of any MATLAB function because it is defined as a Stateflow data of type `ml`.

In the second line of the example, `mldata2`, also of Stateflow data type `ml`, receives the transpose matrix of the matrix in `mldata1`. It is assigned the return value of the MATLAB function `transpose` in which `mldata1` is the argument.

Note the differences in notation if the preceding example were to use MATLAB workspace data (`wsdata1` and `wsdata2`) instead of Stateflow `ml` data to hold the generated matrices:

```
ml.wsdata1 = ml.rand(3);
ml.wsdata2 = ml.transpose(ml.wsdata);
```

In this case, each workspace data must be accessed through the `ml` namespace operator.

Inferring Return Size for ml Expressions

Stateflow expressions using the `ml` namespace operator and the `ml` function are evaluated in the MATLAB workspace at run-time. This means that the actual size of the data returned from the following expression types is known only at run-time:

- MATLAB workspace data or functions using the `ml` namespace operator or the `ml` function call

  For example, the size of the return values from the expressions `ml.var`, `ml.func()`, or `ml(evalString, arg1, arg2,...)`, where `var` is a MATLAB

workspace variable and `func` is a MATLAB function, cannot be known until run-time.

- Stateflow data of type `ml`
- Graphical functions that return Stateflow data of type `ml`

When any of these expressions is used in action language, Stateflow code generation must create temporary Stateflow data, invisible to the user, to hold their intermediate returns for evaluation of the full expression of which they are a part. Because the size of these return values is not known till run-time, Stateflow must employ context rules to infer their size for the creation of the temporary data.

During run-time, if the actual returned value from one of these commands differs from the inferred size of the temporary variable chosen to store it, a size mismatch error results. To prevent these run-time errors, use the following guidelines in constructing action language statements with MATLAB commands or `ml` data:

1. The return sizes of MATLAB commands or data in an expression must match the return sizes of peer expressions.

   For example, in the expression `ml.func() * (x + ml.y)`, if `x` is a 3-by-2 matrix, then `ml.func()` and `ml.y` are also assumed to evaluate to 3-by-2 matrices. If either returns a value of different size (other than a scalar), an error results during run-time.

2. Expressions that return a scalar never produce an error.

   You can combine matrices and scalars in larger expressions because MATLAB practices scalar expansion. For example, in the larger expression `ml.x + y`, if `y` is a 3-by-2 matrix and `ml.x` returns a scalar, the resulting value is determined by adding the scalar value of `ml.x` to every member of `y` to produce a matrix with the size of `y`, that is, a 3-by-2. The same rule applies to subtraction (`-`), multiplication (`*`), division (`/`), and any other binary operations.

3. MATLAB commands or Stateflow data of type `ml` can be members of the following independent levels of expression, for which the return size must be resolved:

- Arguments

  The expression for each function argument is a larger expression for which the return size of MATLAB commands or Stateflow data of type `ml` must be determined. For example, in the expression `z + func(x + ml.y)`, the size of `ml.y` has nothing to do with the size of `z`, because `ml.y` is used at the function argument level. However, the return size for `func(x + ml.y)` must match the size of `z`, because they are both at the same expression level.

- Array indices

  The expression for an array index is an independent level of expression that is required to be scalar in size. For example, in the expression `x + arr[y]`, the size of `y` has nothing to do with the size of `x` because `y` and `x` are at different levels of expression, and `y` must be a scalar.

4 The return size for an indexed array element access must be a scalar.

For example, the expression `x[1][1]`, where `x` is a 3-by-2 array, must evaluate to a scalar.

5 MATLAB command or data elements used in an expression for the input argument for a MATLAB function called through the `ml` namespace operator are resolved for size using the rule for peer expressions (preceding rule 1) for the expression itself, because there is no size definition prototype available.

For example, in the function call `ml.func(x + ml.y)`, if `x` is a 3-by-2 array, `ml.y` must return a 3-by-2 array or a scalar.

6 MATLAB command or data elements used for the input argument for a graphical function in an expression are resolved for size by the function's prototype.

For example, if the graphical function `gfunc` has the prototype `gfunc(arg1)`, where `arg1` is a 2-by-3 Stateflow data array, then the calling expression, `gfunc(ml.y + x)`, requires that both `ml.y` and `x` evaluate to 2-by-3 arrays (or scalars) during run-time.

Note Currently, Stateflow supports only scalar and `ml` type graphical function arguments and return types.

7. `ml` function calls can take only scalar or string literal arguments. Any MATLAB command or data used to specify an argument for the `ml` function must return a scalar value.

8. In an assignment expression, the size of the right-hand expression must match the size of the left-hand expression, with one exception: if the left-hand expression is a single MATLAB variable such as `ml.x` or a single Stateflow data of type `ml`, then the sizes of both left-hand expression and right-hand expression are determined by the right-hand expression.

   For example, in the expression `s = ml.func(x)`, where `x` is a 3-by-2 matrix and `s` is scalar data in Stateflow, `ml.func(X)` must return a scalar to match the left-hand expression, `s`. However, in the expression `ml.y = x + s`, where `x` is a 3-by-2 data array and `s` is scalar, the left-hand expression, workspace variable `y`, is assigned the size of a 3-by-2 array to match the size of the right-hand expression, `x+s`, a 3-by-2 array.

9. If you cannot resolve the return size of MATLAB command or data elements in a larger expression by any of the preceding rules, they are assumed to return scalar values.

   For example, in the expression `ml.x = ml.y + ml.z`, none of the preceding rules can be used to infer a common size among `ml.x`, `ml.y`, and `ml.z`. In this case, both `ml.y` and `ml.z` are assumed to return scalar values. And even if `ml.y` and `ml.z` return matching sizes at run-time, if they return nonscalar values, a size mismatch error results.

10. The preceding rules for resolving the size of member MATLAB commands or Stateflow data of type `ml` in a larger expression apply only to cases in which numeric values are expected for that member. For nonnumeric returns, a run-time error results.

    For example, the expression `x + ml.str`, where `ml.str` is a string workspace variable, produces a run-time error stating that `ml.str` is not a numeric type.

Note Member MATLAB commands or data of type `ml` in a larger expression are limited to numeric values (scalar or array) only if they participate in numeric expressions.

11 Stateflow has special cases in which it does no size checking to resolve the size of MATLAB command or data expressions that are members of larger expressions. In the cases shown, use of a singular MATLAB element such as `ml.var`, `ml.func()`, `ml(evalString, arg1, arg2,...)`, Stateflow data of type `ml`, or a graphical function returning a Stateflow data of type `ml`, does not require that size checking be enforced at run-time. In these cases, assignment of a return to the left-hand side of an assignment statement or to a function argument is made without consideration for a size mismatch between the two:

- An assignment in which the left-hand side is a MATLAB workspace variable

  For example, in the expression `ml.x = ml.y`, `ml.y` is a MATLAB workspace variable of any size and type (structure, cell array, string, and so on).

- `An` assignment in which the left-hand side is a data of type `ml`

  For example, in the expression `m_x = ml.func()`, `m_x` is a Stateflow data of type `ml`.

- Input arguments of a MATLAB function

  For example, in the expression `ml.func(m_x, ml.x, gfunc())`, `m_x` is a Stateflow data of type `ml`, `ml.x` is a MATLAB workspace variable of any size and type, and `gfunc()` is a Stateflow graphical function that returns a Stateflow data of type `ml`. Even though Stateflow does nothing to check the size of the input type, if the passed-in data is not of the expected type, an error results from the function call `ml.func()`.

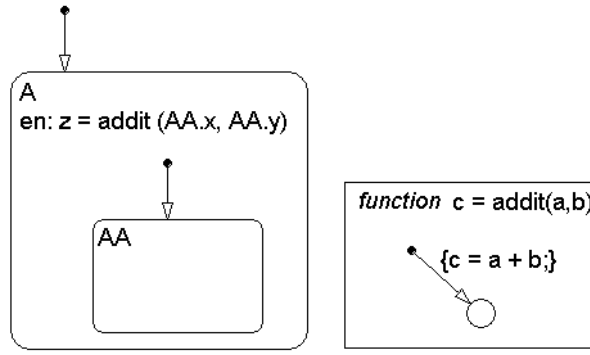
- Arguments for a graphical function that are specified as Stateflow data of type `ml` in its prototype statement

Note If inputs in the preceding cases are replaced with non-MATLAB numeric Stateflow data, a conversion to a `ml` type is performed.

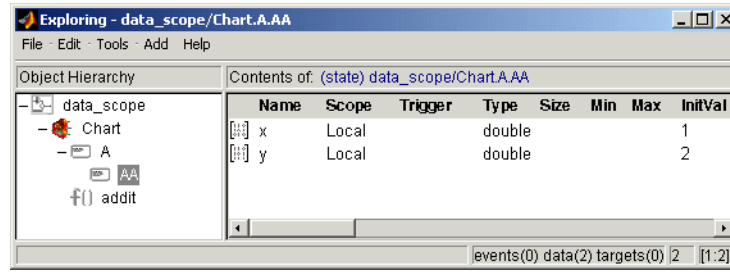
Data and Event Arguments

When you use data and event objects as arguments to functions that you call in action language, they are assumed to be defined at the same level in the hierarchy as the action language that references them. If they are not found at that level, Stateflow attempts to resolve the object name by searching up the hierarchy. Data or event object arguments that are parented anywhere else must have their path hierarchies defined explicitly.

In the following example, state A calls the graphical function `addit` to add the Stateflow data `x` and `y` and store the result in data `z`.



The following Explorer window shows that data `x` and `y` are defined for state AA and not for state A. However, data value z is defined for state A (not shown).



Because state AA owns the data `x` and `y`, when state A passes them as arguments to the graphical function `addit`, they must be referenced by their path hierarchy.

There are a variety of functions that you can call in Stateflow action language that use data as arguments. See the following sections:

- "Using Graphical Functions in Stateflow Charts" on page 5-48
- "Calling C Functions" on page 7-42
- "MATLAB Functions and Data" on page 7-46

Only the temporal logic operators take events as an argument. See "Using Temporal Logic" on page 7-68.

Arrays in Action Language

This section tells you how to use Stateflow data arrays in action language. See the following topics:

- "Array Notation" on page 7-60 — Gives examples of array notation in action language.
- "Arrays and Custom Code" on page 7-61 — Shows you how to access arrays provided by custom code that you build into a Stateflow target.

Array Notation

Use C style syntax in the action language to access array elements.

```
local_array[1][8][0] = 10;

local_array[i][j][k] = 77;

var = local_array[i][j][k];
```

As an exception to this style, **scalar expansion** is available within the action language. This statement assigns a value of 10 to all the elements of the array `local_array`.

```
local_array = 10;
```

Scalar expansion is available for performing general operations. This statement is valid if the arrays `array_1`, `array_2`, and `array_3` have the same value for the **Sizes** property.

```
array_1 = (3*array_2) + array_3;
```

Note Use the same notation for accessing arrays in Stateflow, from Simulink, and from custom code.

Arrays and Custom Code

Stateflow action language provides the same syntax for Stateflow arrays and custom code arrays.

See also “Integrating Custom Code with Stateflow Diagrams” on page 11–20.

Note Any array variable that is referred to in a Stateflow chart but is not defined in the data dictionary is identified as a custom code variable.

Event Broadcasting

You can specify an event to be broadcast in the action language. Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event's parent that determines who can trigger on the event (has receive rights). See "Name" on page 6-5 for more information.

See the following sections for an understanding of broadcasting events in action language:

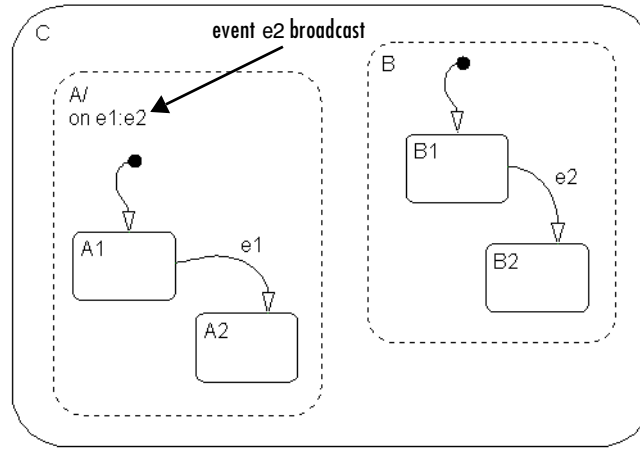
- "Broadcasting Events" on page 7-62 — Gives examples of using broadcast events to synchronize behavior between AND (parallel) states.
- "Directed Event Broadcasting" on page 7-64 — Shows you how to use the send function to send an event to a specific state.

Broadcasting Events

Broadcasting an event in the action language is most useful as a means of synchronization among AND (parallel) states. Recursive event broadcasts can lead to definition of cyclic behavior. Cyclic behavior can be detected only during simulation.

Event Broadcast State Action Example

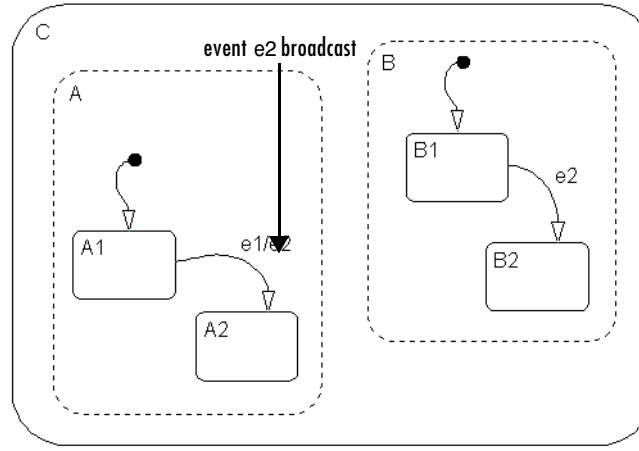
The following is an example of the state action notation for an event broadcast:



See “Event Broadcast State Action Example” on page 4-65 for information on the semantics of this notation.

Event Broadcast Transition Action Example

The following is an example of transition action notation for an event broadcast.



See “Event Broadcast Transition Action with a Nested Event Broadcast Example” on page 4-69 for information on the semantics of this notation.

Directed Event Broadcasting

You can specify a directed event broadcast in the action language. Using a directed event broadcast, you can broadcast a specific event to a specific receiver state. Directed event broadcasting is a more efficient means of synchronization among parallel (AND) states. Using directed event broadcasting improves the efficiency of the generated code. As is true in event broadcasting, recursive event broadcasts can lead to definition of cyclic behavior.

Note An action in one chart cannot broadcast events to states defined in another chart.

The format of the directed broadcast is

```
send(event_name,state_name)
```

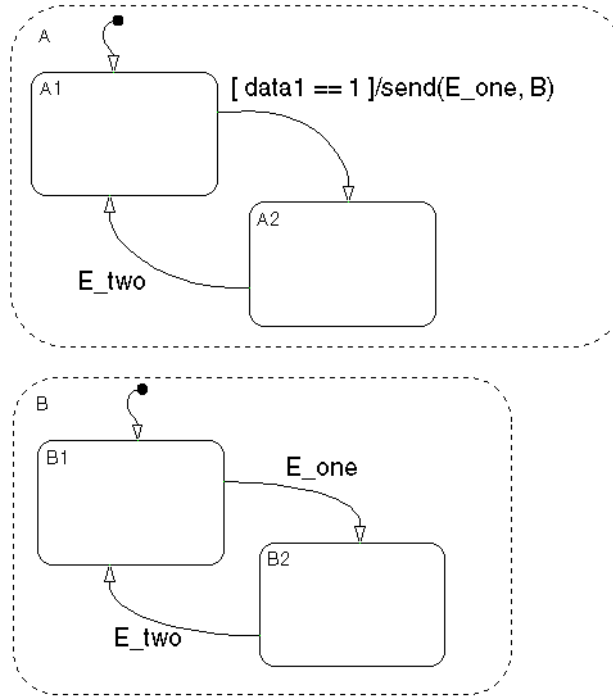
where `event_name` is broadcast to `state_name` (and any offspring of that state in the hierarchy). The `state_name` argument can include a full hierarchy path. For example,

```
send(event_name, chart_name.state_name1.state_name2)
```

The `state_name` specified must be active at the time the `send` is executed for the `state_name` to receive and potentially act on the directed event broadcast.

Directed Event Broadcast Using send Example

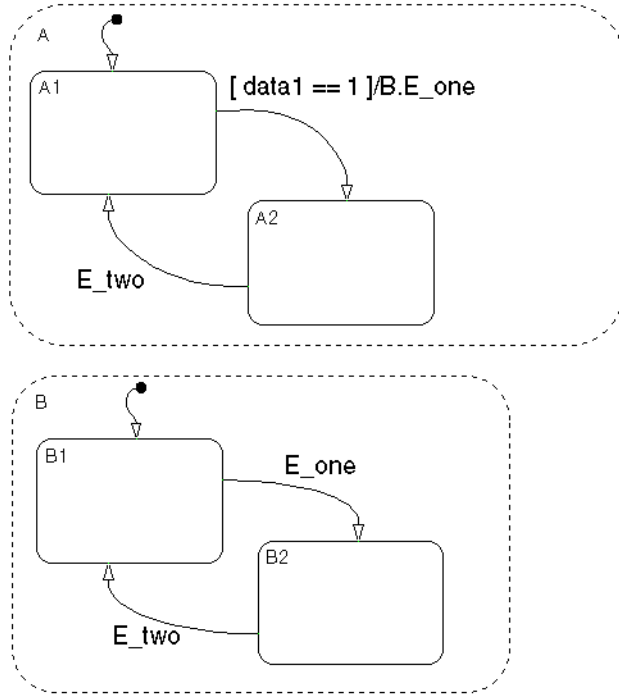
This is an example of a directed event broadcast using the `send(event_name,state_name)` transition action as a transition action.



In this example, event `E_one` must be visible in both `A` and `B`. See "Directed Event Broadcasting Using Qualified Event Names Example" on page 4-79 for information on the semantics of this notation.

Directed Event Broadcast Using Qualified Event Names Example

This example illustrates the use of a qualified event name in the event broadcast of the previous example.



See “Directed Event Broadcasting Using Qualified Event Names Example” on page 4-79 for information on the semantics of this notation.

Condition Statements

You sometimes want transitions or actions associated with transitions to take place only if a given condition is true. Conditions are placed within square brackets ([ ]). The following are some guidelines for defining and using conditions:

- The condition expression must be a Boolean expression of some kind that evaluates to either true (1) or false (0).
- The condition expression can consist of any of the following:
  - Boolean operators that make comparisons between data and numeric values
  - A function that returns a Boolean value
  - The `In(state_name)` condition function that is evaluated as true when the state specified as the argument is active. The full state name, including any ancestor states, must be specified to avoid ambiguity.

Note A chart cannot use the `In` condition function to trigger actions based on the activity of states in other charts.

  - Temporal conditions (see "Using Temporal Logic" on page 7-68)
- The condition expression should not call a function that causes the Stateflow diagram to change state or modify any variables.
- Boolean expressions can be grouped using & for expressions with AND relationships and | for expressions with OR relationships.
- Assignment statements are not valid condition expressions.
- Unary increment and decrement actions are not valid condition expressions.

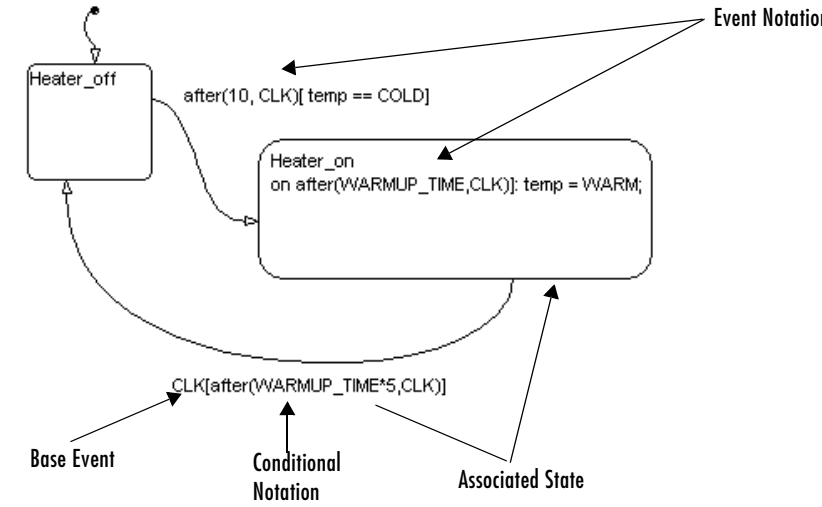
Using Temporal Logic

Temporal logic operators are Boolean operators that operate on recurrence counts of Stateflow events. See the following subsections for individual descriptions of each temporal logic operator:

- "Rules for Using Temporal Logic Operators" on page 7-68 — Gives you the rules for using temporal logic operators.
- "after Temporal Logic Operator" on page 7-70 — Shows you how to use the after temporal logic operator.
- "before Temporal Logic Operator" on page 7-71 — Shows you how to use the before temporal logic operator.
- "at Temporal Logic Operator" on page 7-72 — Shows you how to use the at temporal logic operator.
- "every Temporal Logic Operator" on page 7-73 — Shows you how to use the every temporal logic operator.
- "Conditional and Event Notation" on page 7-74 — Describes the two types of notation you can use for temporal logic operations in action language.
- "Temporal Logic Events" on page 7-74 — Shows you how you temporal logic is a single event equivalent to accumulated base events.

Rules for Using Temporal Logic Operators

The following diagram illustrates the use of temporal logic operators in action language:



The following rules apply generally to the use of temporal logic operators:

- The recurring event on which a temporal operator operates is called the *base event*. Any Stateflow event can serve as a base event for a temporal operator.

Note Temporal logic operators can also operate on recurrences of implicit events, such as state `entry` or `exit` events or data `change` events. See "Keywords in Action Language" on page 7-4.

- For a chart with no Simulink input events, you can use the `wakeup` (or `tick`) event to denote the implicit event of a chart waking up.
- Temporal logic operators can appear only in conditions on transitions originating from states and in state actions.

Note This means you cannot use temporal logic operators as conditions on default transitions or flow graph transitions.

The state on which the temporally conditioned transition originates or in whose `during` action the condition appears is called the temporal operator's *associated state*.

- You must use event notation (see "Temporal Logic Events" on page 7-74) to express temporal logic conditions on events in state `during` actions.

after Temporal Logic Operator

The syntax for the `after` operator is as follows:

```
after(n, E)
```

where `E` is the base event for the `after` operator and `n` is one of the following:

- A constant integer greater than 0
- An expression that evaluates to an integer value greater than or equal to 0

The `after` operator is true if the base event `E` has occurred `n` times since activation of its associated state. Otherwise, it is false. In a chart with no input events, `after(n,wakeup)` (or `after(n,tick)`) evaluates to true after the chart has woken up `n` times.

Note The `after` operator resets its counter for `E` to 0 each time the associated state is activated.

The following example illustrates use of the `after` operator in a transition expression.

```
CLK[after(10, CLK) && temp == COLD]
```

This example permits a transition out of the associated state only if there have been 10 occurrences of the `CLK` event since the state was activated and the `temp` data item has the value `COLD`.

The next example illustrates usage of event notation for temporal logic conditions in transition expressions.

```
after(10, CLK)[temp == COLD]
```

This example is semantically equivalent to the first example.

The next example illustrates setting a transition condition for any event visible in the associated state while it is activated.

```
[after(10, CLK)]
```

This example permits a transition out of the associated state on any event after 10 occurrences of the `CLK` event since activation of the state.

The next two examples underscore the semantic distinction between an `after` condition on its own base event and an `after` condition on a nonbase event.

```
CLK[after(10,CLK)]
ROTATE[after(10,CLK]
```

The first expression says that the transition must occur *as soon as* 10 `CLK` events have occurred after activation of the associated state. The second expression says that the transition can occur *no sooner than* 10 `CLK` events after activation of the state, but possibly later, depending on when the `ROTATION` event occurs.

The next example illustrates usage of an `after` event in a state's `during` action.

```
Heater_on
on after(5*BASE_DELAY, CLK): status('heater on');
```

This example causes the `Heater_on` state to display a status message each `CLK` cycle, starting `5*BASE_DELAY` clock cycles after activation of the state. Note the use of event notation to express the after condition in this example. Use of conditional notation is not allowed in state `during` actions.

before Temporal Logic Operator

The before operator has the following syntax:

```
before(n, E)
```

where `E` is the base event for the `before` operator and `n` is one of the following:

- A constant integer greater than 0

- An expression that evaluates to an integer value greater than or equal to 0

The `before` operator is true if the base event `E` has occurred less than `n` times since activation of its associated state. Otherwise, it is false. In a chart with no input events, `before(n,wakeup)` or `before(n,tick)` evaluates to true before the chart has woken up `n` times.

Note The `before` operator resets its counter for `E` to 0 each time the associated state is activated.

The following example illustrates the use of the `before` operator in a transition expression.

```
ROTATION[before(10, CLK)]
```

This expression permits a transition out of the associated state only on occurrence of a `ROTATION` event but *no later than* `10 CLK` cycles after activation of the state.

The next example illustrates usage of a `before` event in a state's `during` action.

```
Heater_on
on before(MAX_ON_TIME, CLK): temp++;
```

This example causes the `Heater_on` state to increment the `temp` variable once per `CLK` cycle until the `MAX_ON_TIME` limit is reached.

at Temporal Logic Operator

The syntax for the `at` operator is as follows:

```
at(n, E)
```

where `E` is the base event for the `at` operator and `n` is one of the following:

- A constant integer greater than 0
- An expression that evaluates to an integer value greater than or equal to 0

The `at` operator is true only at the n^{th} occurrence of the base event `E` since activation of its associated state. Otherwise, it is false. In a chart with no input events, `at(n,wakeup)` (or `at(n,tick)`) evaluates to true when the chart wakes up for the n^{th} time.

Note The `at` operator resets its counter for `E` to 0 each time the associated state is activated.

The following example illustrates the use of the `at` operator in a transition expression.

```
ROTATION[at(10, CLK)]
```

This expression permits a transition out of the associated state only if a `ROTATION` event occurs *exactly* `10 CLK` cycles after activation of the state.

The next example illustrates usage of an `at` event in a state's `during` action.

```
Heater_on
on at(10, CLK): status('heater on');
```

This example causes the `Heater_on` state to display a status message 10 `CLK` cycles after activation of the associated state.

every Temporal Logic Operator

The syntax for the `every` operator is as follows:

```
every(n, E)
```

where `E` is the base event for the `every` operator and `n` is one of the following:

- A constant integer greater than 0
- An expression that evaluates to an integer value greater than or equal to 0

The `every` operator is true at every `n`th occurrence of the base event `E` since activation of its associated state. Otherwise, it is false. In a chart with no input events, `every(n,wakeup)` (or `every(n,tick)`) evaluates to true whenever the chart wakes up an integer multiple `n` times.

Note The `every` operator resets its counter for `E` to 0 each time the associated state is activated. As a result, this operator is useful only in state `during` actions.

The following example illustrates the use of the `every` operator in a state.

```
Heater_on
on every(10, CLK): status('heater on');
```

This example causes the `Heater_on` state to display a status message every 10 `CLK` cycles after activation of the associated state.

Conditional and Event Notation

Stateflow treats the following notations as equivalent,

```
E[tlo(n, E) && C]
tlo(n, E)[C]
```

where *`tlo`* is a temporal logic operator (`after`, `before`, `at`, `every`), `E` is the operator's base event, `n` is the operator's occurrence count, and `C` is any conditional expression. For example, the following expressions are functionally equivalent in Stateflow:

```
CLK[after(10, CLK) && temp == COLD]
after(10, CLK)[temp == COLD]
```

The first notation is referred to as the conditional notation for temporal logic operators and the second notation as the event notation.

Note You can use conditional and event notation interchangeably in transition expressions. However, you must use the event notation in state `during` actions.

Temporal Logic Events

Although temporal logic does not introduce any new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. For example, suppose that you want a transition to occur from state `A` exactly 10 clock cycles after activation of the state. One way to achieve this would be to define an event called `ALARM` and to broadcast this event `10` `CLK` events after state `A` is entered. You would then use `ALARM` as the event that triggers the transition out of state `A`.

An easier way to achieve the same behavior is to set a temporal logic condition on the CLK event that triggers the transition out of state A.

```
CLK[after(10, CLK)]
```

Note that this approach does not require creation of any new events. Nevertheless, conceptually it is useful to think of this expression as equivalent to creation of an implicit event that triggers the transition. Hence, Stateflow supports the equivalent event notation (see "Temporal Logic Events" on page 7-74).

```
after(10, CLK)
```

Note that the event notation allows you to set additional constraints on the implicit temporal logic "event," for example,

```
after(10, CLK)[temp == COLD]
```

This expression says, "Exit state `A` if the temperature is cold but no sooner than `10` clock cycles."

Special Symbols

Stateflow notation uses the symbols `t`, `$`, `...`, `%`, `//`, `/*`, `;`, `F`, and hexadecimal notation to provide the user with special features in action language notation. These uses are described in the topics that follow.

Time Symbol 't'

You can use the letter `t` to represent absolute time in simulation targets. This simulation time is inherited from Simulink.

For example, the condition `[t - On_time > Duration]` specifies that the condition is true if the value of `On_time` subtracted from the simulation time `t` is greater than the value of `Duration`.

The meaning of `t` for nonsimulation targets is undefined since it is dependent upon the specific application and target hardware.

Literal Code Symbol '$'

Place action language you want the parser to ignore but you want to appear as entered in the generated code within `$` characters. For example,

```
$
ptr -> field = 1.0;
$
```

The parser is completely disabled during the processing of anything between the `$` characters. Frequent use of literals is discouraged.

Continuation Symbol '...'

Enter the characters `...` at the end of a line to indicate the expression continues on the next line.

Comment Symbols %, //, and /*

Stateflow action language supports the following comment formats:

- `%` MATLAB comment line
- `//` C++ comment line
- /* C comment line */

MATLAB Display Symbol ';'

Omitting the semicolon after an expression displays the results of the expression in the MATLAB Command Window. If you use a semicolon, the results are not displayed.

Single-Precision Floating-Point Number Symbol F

Stateflow action language recognizes a trailing “F” for specifying single-precision floating-point numbers as in the action statement x = 4.56F;. If a trailing “F” does not appear with a number, it is assumed to be double-precision.

Hexadecimal Notation

The action language supports C style hexadecimal notation (for example, `0xFF`). You can use hexadecimal values wherever you can use decimal values.

8

Defining Stateflow Interfaces

Each Stateflow chart is a block in a Simulink diagram that sits on top of MATLAB. You can share data with MATLAB and Simulink and also determine how and when Simulink executes your charts through Stateflow interfaces with the following sections:

Overview Stateflow Interfaces (p. 8-2)	Each Stateflow block interfaces to its Simulink model. Take an overview look at Stateflow interfaces to Simulink and MATLAB.
Setting the Stateflow Block Update Method (p. 8-4)	Implementing different Stateflow interfaces in Simulink requires you to set the update method for your chart. This section describes each of the settings for the update method of your chart.
Adding Input or Output Data and Events to Charts (p. 8-6)	Implementing different Stateflow interfaces in Simulink requires you to add input or output data or events to your chart. This section describes how you add input or output data or events to your chart.
Implementing Interfaces in Stateflow to Simulink (p. 8-11)	This section summarizes all the settings necessary for implementing any possible interface in Simulink to your Stateflow chart block.
MATLAB Workspace Interfaces (p. 8-22)	The MATLAB workspace is an area of memory normally accessible from the MATLAB command line. This section describes ways that Stateflow can access the data and functions of the MATLAB workspace.
Interface to External Sources (p. 8-23)	Describes the ways in which a Stateflow chart can interface data and events outside its Simulink model.

Overview Stateflow Interfaces

Each Stateflow block interfaces to its Simulink model. Take an overview look at Stateflow interfaces to Simulink and MATLAB with the following topics:

- "Stateflow Interfaces" on page 8-2 — Lists the interfaces that Stateflow has to Simulink blocks, MATLAB data, and external code sources.
- "Typical Tasks to Define Stateflow Interfaces" on page 8-3 — Describes the tasks used to define Stateflow interfaces.
- "Where to Find More Information on Events and Data" on page 8-3 — Gives you references to further information on defining Stateflow interfaces in the Stateflow documentation.

Stateflow Interfaces

Each Stateflow block interfaces to its Simulink model. Each Stateflow block can interface to sources external to the Simulink model (data, events, custom code). Events and data are the Stateflow objects that define the interface from the Stateflow block's point of view.

Events can be local to the Stateflow block or can be propagated to and from Simulink and sources external to Simulink. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to sources external to the Simulink model.

The Stateflow interfaces includes the following:

- Physical connections between Simulink blocks and the Stateflow block
- Event and data information exchanged between the Stateflow block and external sources
- Graphical functions exported from a chart

  See "Exporting Graphical Functions" on page 5-53 for more details.
- The MATLAB workspace

  See "MATLAB Functions and Data" on page 7-46 for more details.
- Definitions in external code sources

Typical Tasks to Define Stateflow Interfaces

Defining the interface for a Stateflow block in a Simulink model involves some or all the tasks described in the following topics:

- Specify the update method for a Stateflow block in a Simulink model.

  This task is described in "Setting the Stateflow Block Update Method" on page 8-4.
- Define the **Output to Simulink** and **Input from Simulink** data and events that you need.

  See the topic "Adding Input or Output Data and Events to Charts" on page 8-6.
- Add and define any nonlocal data and events your Stateflow diagram must interact with.
- Define relationships with any external sources.

  See the topics "MATLAB Workspace Interfaces" on page 8-22 and "Interface to External Sources" on page 8-23.

The precediing task list could be a typical sequence. You might find that another sequence better complements your model development.

See "Implementing Interfaces in Stateflow to Simulink" on page 8-11 for examples of implemented interfaces to Simulink.

Where to Find More Information on Events and Data

The following references are relevant to defining the interface for a Stateflow Chart block in Simulink:

- "Defining Input Events" on page 6-8
- "Defining Output Events" on page 6-9
- "Importing Events" on page 6-10
- "Exporting Events" on page 6-9
- "Defining Input Data" on page 6-26
- "Defining Output Data" on page 6-27
- "Importing Data" on page 6-29
- "Exporting Data" on page 6-28

Setting the Stateflow Block Update Method

Implementing different Stateflow interfaces in Simulink requires you to set the update method for your chart. This section describes each of the settings for the update method of your chart.

Setting the update method for your Stateflow chart block is only part of the process of implementing an interface to your Stateflow block in Simulink. To see a summary of the settings necessary for implementing any possible interfaces in Simulink to your Stateflow chart block, see "Implementing Interfaces in Stateflow to Simulink" on page 8-11.

Stateflow Block Update Methods

Stateflow blocks are Simulink subsystems. Simulink events wake up subsystems for execution. To specify a wakeup method for a chart, set the chart's **Update method** property in the properties dialog for the chart (see "Specifying Chart Properties" on page 5-78). Choose from the following wakeup methods:

- **Triggered/Inherited**

  This is the default update method. Specifying this method causes input from the Simulink model to determine when the chart wakes up during a simulation.

  - Triggered

    If you define input events for the chart, the Stateflow block is explicitly triggered by a signal on its trigger port originating from a connected Simulink block. This trigger input event can be set in the Stateflow Explorer to occur in response to a Simulink signal that is **Rising**, **Falling**, or **Either** (rising and falling), or in response to a **Function Call**. See "Defining Input Events" on page 6-8 and "Adding Input Events from Simulink" on page 8-6.

  - Inherited

    If you do not define input events, the Stateflow block implicitly inherits triggers from the Simulink model. These implicit events are the sample times (discrete-time or continuous) of the Simulink signals providing inputs to the chart. If you define data inputs (see "Defining Input Data" on page 6-26), the chart awakens at the rate of the fastest data input. If you

do not define any data inputs for the chart, the chart wakes up as defined by its parent subsystem's execution behavior.

- **Sampled**

  Simulink awakens (samples) the Stateflow block at the rate you specify as the block's **Sample Time** property. An implicit event is generated by Simulink at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the Simulink model can have different sample times.

- **Continuous**

  Simulink wakes up (samples) the Stateflow block at each step in the simulation, as well as at intermediate time points that can be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

See "Interface to External Sources" on page 8-23 for more information.

Adding Input or Output Data and Events to Charts

Implementing different Stateflow interfaces in Simulink requires you to add input or output data or events to your chart. This section describes how you add input or output data or events to your chart in the following sections:

- "Adding Input Events from Simulink" on page 8-6 — Tells you how to add an input event to a chart in Stateflow Explorer.
- "Adding Output Events to Simulink" on page 8-7 — Tells you how to add and define the necessary fields for an event output to Simulink.
- "Adding Input Data from Simulink" on page 8-8 — Tells you how to add a data input from Simulink to a Stateflow chart in Stateflow Explorer.
- "Adding Output Data to Simulink" on page 8-9 — Tells you how to add and define the necessary fields for a data output to Simulink.

Setting the update method for your Stateflow chart block is only part of the process of implementing an interface to your Stateflow block in Simulink. To see a summary of the settings necessary for implementing any possible interfaces in Simulink to your Stateflow chart block, see "Implementing Interfaces in Stateflow to Simulink" on page 8-11.

Adding Input Events from Simulink

The following steps describe how to add an input event to a chart in Stateflow Explorer:

1 Add an event to the chart using the Stateflow diagram editor or Stateflow Explorer (see "Adding Events to the Data Dictionary" on page 6-2).

   You must add the event to the chart and not to any other object in the chart. Adding events with the Stateflow diagram editor adds them only to the chart that is visible in the diagram editor. In Stateflow Explorer, you must select (highlight) the chart object in the hierarchy shown in the left pane of the Explorer before adding the event.

2 Set the event's **Scope** to **Input from Simulink**.

   A single Simulink trigger port is added to the top of the Stateflow block.

3 Change the event's **Trigger** for the input event from the resulting pop-up menu.

   The type of **Trigger** indicates what kind of signal in Simulink triggers the input event to the chart. It can have one of the following values:

Trigger	Description
Rising	Rising edge trigger, where the control signal changes from either 0 or a negative value to a positive value.
Falling	Falling edge trigger, where the control signal changes from either 0 or a positive value to a negative value.
Either	Either rising or falling edge trigger.
Function Call	Function call triggered.

   See "Specifying Trigger Types" on page 6-11 for more information.

If you choose to add an event in the Stateflow Explorer, you can also change its Name, Scope, and Trigger properties directly in the Explorer (see "Setting Properties for Data, Events, and Targets" on page 10-9).

Adding Output Events to Simulink

The following steps describe how to add and define the necessary fields for an event output to Simulink:

1 Add an event to the chart using the Stateflow diagram editor or Stateflow Explorer (see "Adding Data to the Data Dictionary" on page 6-14).

   You must add the event to the chart and not to any other object in the chart. Adding an event with the Stateflow diagram editor adds it only to the chart that is visible in the diagram editor. In Stateflow Explorer, you must select (highlight) the chart object in the hierarchy shown in the left pane of the Explorer before adding the event.

2 Set the **Scope** field to **Output to Simulink**.

When you define an event to be **Output to Simulink**, a Simulink output port is added to the Stateflow block. Output events from the Stateflow block to the Simulink model are scalar.

If you choose to add an event in the Stateflow Explorer, you can also change its **Name** and **Scope** properties directly in the Explorer (see "Setting Properties for Data, Events, and Targets" on page 10-9).

Adding Input Data from Simulink

The following steps describe how to add a data input from Simulink to a Stateflow chart in Stateflow Explorer:

1 Add a data object to the chart using the Stateflow diagram editor or Stateflow Explorer (see "Adding Data to the Data Dictionary" on page 6-14).

You must add the data to the chart and not to any other object in the chart. Adding a data object with the Stateflow diagram editor adds it only to the chart that is visible in the diagram editor. In Stateflow Explorer, you must select (highlight) the chart object in the hierarchy shown in the left pane of the Explorer before adding the event.

2 Set the data object's **Scope** to **Input from Simulink** from the resulting menu.

A single Simulink data input port is added to the side of the Stateflow block. When you add a data input, each data input is represented on the Stateflow block by a Simulink input port. Multiple data inputs to the Stateflow block must be scalar (they cannot be vectorized).

3 Specify the remaining properties for the data.

Data input to Stateflow can be a scalar, or it can be a vector or two-dimensional matrix array of fixed size. It is not possible to change the array size during run-time. See "Defining Data Arrays" on page 6-24.

If you choose to add data in the Stateflow Explorer, you can also change some of its properties directly in the Explorer (see "Setting Properties for Data, Events, and Targets" on page 10-9).

Note If you want the input port corresponding to this input data item to accept Simulink data of type other than `double`, you must select the chart's strong data typing option. See "Defining Input Data" on page 6-26 and "Specifying Chart Properties" on page 5-78 for more information.

Adding Output Data to Simulink

The following steps describe how to add and define the necessary fields for a data output to Simulink:

1. Add a data object to the chart using the Stateflow diagram editor or Stateflow Explorer (see "Adding Data to the Data Dictionary" on page 6-14).

   You must add the data to the chart and not to any other object in the chart. Adding a data object with the Stateflow diagram editor adds it only to the chart that is visible in the diagram editor. In Stateflow Explorer, you must select (highlight) the chart object in the hierarchy shown in the left pane of the Explorer before adding the event.

2. Set the **Scope** of the data to **Output to Simulink**.

   When you define a data object to be **Output to Simulink**, a Simulink output port is added to the Stateflow block. Output data objects from the Stateflow block to the Simulink model are scalar.

Specify the remaining properties for the data. Data output to Stateflow can be a scalar, or it can be a vector or two-dimensional matrix array of fixed size. It is not possible to change the array size at run-time. See "Defining Data Arrays" on page 6-24

If you choose to add data in the Stateflow Explorer, you can also change some of its properties directly in the Explorer (see "Setting Properties for Data, Events, and Targets" on page 10-9).

Note If you want the output port corresponding to this output data item to emit data of type other than `double`, you must select the chart's strong data typing option. See "Defining Input Data" on page 6-26 and "Specifying Chart Properties" on page 5-78 for more information.

Click **Apply** to save the properties. Click **OK** to save the properties and close the dialog box.

Implementing Interfaces in Stateflow to Simulink

This section summarizes all the settings necessary for implementing any of the following possible interfaces to Simulink in your Stateflow chart block:

- "Defining a Triggered Stateflow Block" — Provides an example of a triggered Stateflow block in a Simulink model.
- "Defining a Sampled Stateflow Block" — Provides an example of a sampled Stateflow block in a Simulink model.
- "Defining an Inherited Stateflow Block" — Provides an example of a Stateflow block that inherits its sample time in a Simulink model.
- "Defining a Continuous Stateflow Block" — Provides an example of a continuously sampled Stateflow block in a Simulink model.
- "Defining Function Call Output Events" — Provides an example of a Stateflow block that triggers a subsystem in a Simulink model with a function call.
- "Defining Edge-Triggered Output Events" — Provides an example of a Stateflow block that triggers a subsystem in a Simulink model with a signal edge.

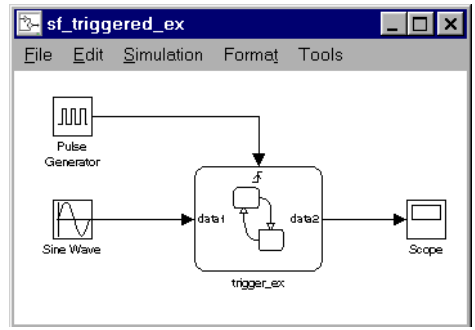
Defining a Triggered Stateflow Block

These are essential conditions that define an edge-triggered Stateflow block:

- The chart **Update method** (set in the **Chart Properties** dialog box) is set to **Triggered or Inherited**. (See "Specifying Chart Properties" on page 5-78.)
- The chart has an **Input from Simulink** event defined and an edge-trigger type specified. (See "Defining Input Events" on page 6-8.)

Triggered Stateflow Block Example

A Pulse Generator block connected to the trigger port of the Stateflow block is an example of an edge-triggered Stateflow block.



The **Input from Simulink** event has a **Rising Edge** trigger type. If more than one **Input from Simulink** event is defined, the sample times are determined by Simulink to be consistent with various rates of all the incoming signals. The outputs of a triggered Stateflow block are held after the execution of the block.

Defining a Sampled Stateflow Block

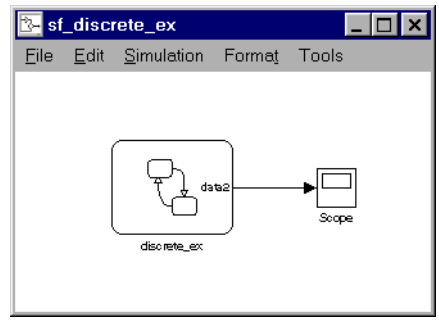
There are two ways you can define a sampled Stateflow block. Setting the chart **Update method** (set in the **Chart Properties** dialog box) to **Sampled** and entering a **Sample Time** value define a sampled Stateflow block. (See "Specifying Chart Properties" on page 5-78.)

Alternatively, you can add and define an **Input from Simulink** data object. Data is added and defined using either the graphics editor **Add** menu or the Explorer. (See "Defining Input Data" on page 6-26.) The chart sample time is determined by Simulink to be consistent with the rate of the incoming data signal.

The **Sample Time** (set in the **Chart Properties** dialog box) takes precedence over the sample time of any **Input from Simulink** data.

Sampled Stateflow Block Example

You specify a discrete sample rate to have Simulink trigger a Stateflow block that is not explicitly triggered via the trigger port. You can specify a **Sample Time** in the Stateflow diagram's **Chart properties** dialog box. The Stateflow block is then called by Simulink at the defined, regular sample times.



The outputs of a sampled Stateflow block are held after the execution of the block.

Defining an Inherited Stateflow Block

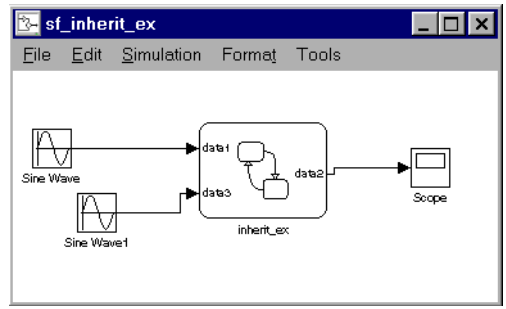
These are essential conditions that define an inherited trigger Stateflow block:

- The chart **Update method** (set in the **Chart Properties** dialog box) is set to **Triggered** or **Inherited**. (See "Specifying Chart Properties" on page 5-78)
- The chart has an **Input from Simulink** data object defined (added and defined using either the graphics editor **Add** menu or the Explorer). (See "Defining Input Data" on page 6-26.) The chart sample time is determined by Simulink to be consistent with the rate of the incoming data signal.

Inherited Stateflow Block Example

Simulink can trigger a Stateflow block that is not explicitly triggered by a trigger port or a specified discrete sample time. In this case, the Stateflow block is called by Simulink at a sample time determined by Simulink.

In this example, more than one **Input from Simulink** data object is defined. The sample times are determined by Simulink to be consistent with the rates of both incoming signals.



The outputs of an inherited trigger Stateflow block are held after the execution of the block.

Defining a Continuous Stateflow Block

To define a continuous Stateflow block, set the chart **Update method** (set in the **Chart Properties** dialog box) to **Continuous**. See "Specifying Chart Properties" on page 5-78.

Considerations in Choosing Continuous Update

The availability of intermediate data makes it possible for the solver to back up in time to precisely locate a zero crossing. Refer to the Using Simulink documentation for further information on zero crossings. Use of the intermediate time point information can provide increased simulation accuracy.

To support the **Continuous** update method, Stateflow keeps an extra copy of all its data.

In most cases, including continuous-time simulations, the **Inherited** method provides consistent results. The timing of state and output changes of the Stateflow block is entirely consistent with that of the continuous plant model.

There are situations, such as the following, when changes within the Stateflow block must be felt immediately by the plant and a **Continuous** update is needed:

- Data output to Simulink that is a direct function of data input from Simulink and then updated by the Stateflow diagram (state **during** actions in particular)

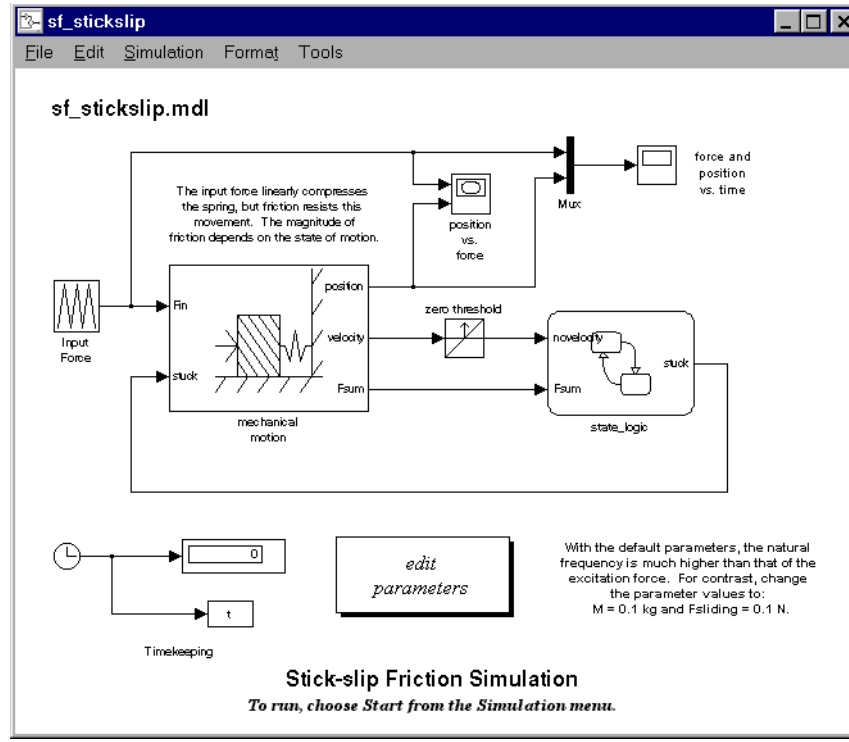
- Models in which Stateflow states correspond to intrinsic physical states, such as the onset of static friction or the polarity of a magnetic domain

  These states are in contrast to states that are assigned, for example, as modes of control strategy.

Continuous Stateflow Block Example

Simulink awakens (samples) the Stateflow block at each step in the simulation, as well as at intermediate time points that might be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

In the following example (provided in the Examples/Stick-Slip Friction Demonstration block), the chart **Update method** (set in the **Chart Properties** dialog box) is set to **Continuous**.



Defining Function Call Output Events

Use a function call output event from a Stateflow block to trigger the execution of a connected Simulink subsystem with the following procedure:

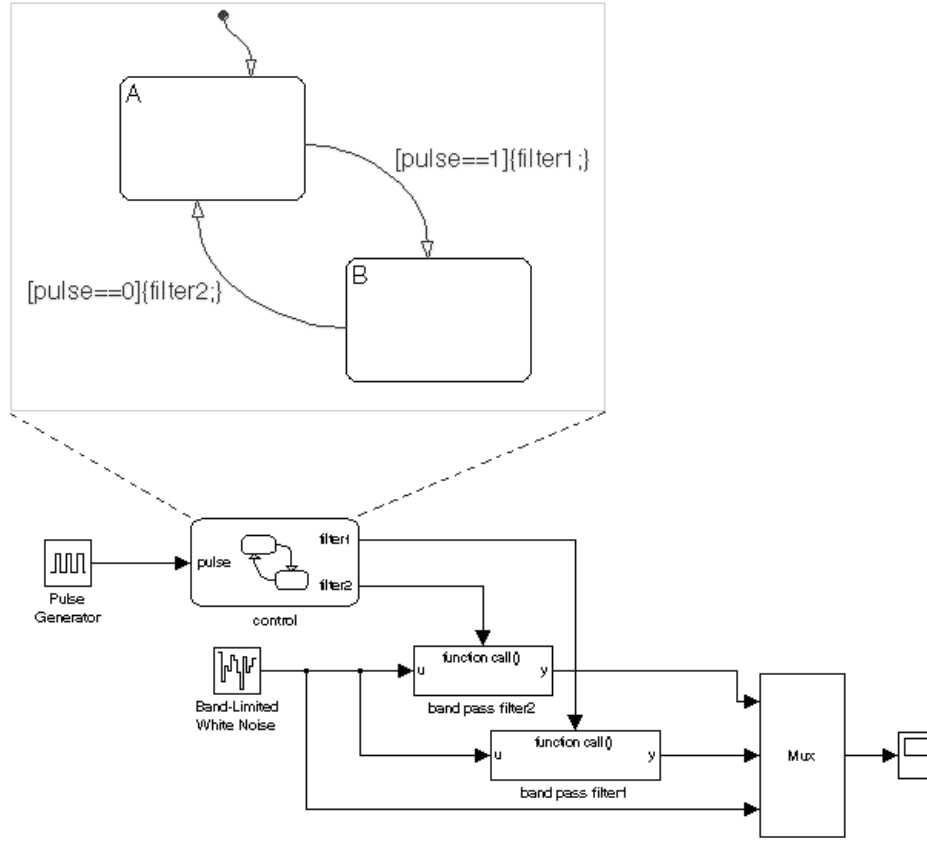
1. Use Stateflow Explorer or the **Add** -> **Data** menu selection in the Stateflow diagram editor to add an **Output to Simulink** event.
2. Select **Function Call** for the event's **Trigger** field the properties dialog for the event. See "Defining Output Events" on page 6-9).
3. In Simulink, place a Trigger block in the subsystem you want to trigger from Stateflow.
4. In the **Triggerport parameters** dialog for the Trigger block, set the **Trigger type** field to **function-call**.
5. Connect the output port on the Stateflow block for the **Function Call** trigger **Output to Simulink** event to the function-call trigger input port of the subsystem.

You should avoid placing any other blocks in the connection lines between the two blocks for Stateflow blocks that have feedback loops from a block triggered by a function call.

Note You cannot connect a function-call output event from Stateflow to a Simulink Demux block in order to trigger multiple subsystems.

Function Call Output Events Example

The following example demonstrates how Stateflow uses function call output events to call Simulink subsystems:



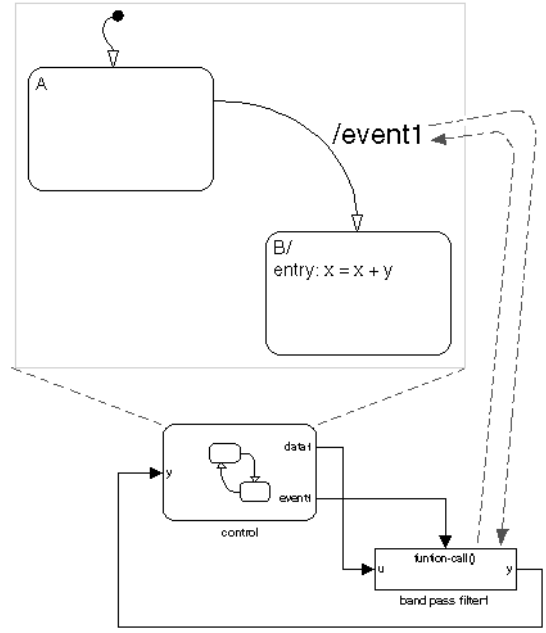
The control Stateflow block has one data input called `pulse` and two event `Function Call` outputs called `filter1` and `filter2`. A pulse generator provides input data to the control block. Each function call output event is attached to a subsystem in the Simulink model that is set to trigger by a function call.

Each transition in the control chart has a condition based on the size of the input pulse. When taken, each transition broadcasts a function call output event that determines whether to make a function call to filter1 or `filter2`. If the **Output to Simulink** function call event `filter1` is broadcast, the `band pass filter1` subsystem executes. If the **Output to Simulink** function call event `filter2` is broadcast, the `band pass filter2` subsystem executes. When

either of these subsystems is finished executing, control is returned to the control Stateflow block for the next execution step. In this way, the Stateflow block control controls the execution of `band pass filter1` and `band pass filter2`.

Function Call Semantics Example

In this example the transition from state `A` to state `B` (in the Stateflow diagram) has a transition action that specifies the broadcast of `event1`. `event1` is defined in Stateflow to be an **Output to Simulink** with a **Function Call** trigger type. The Stateflow block output port for `event1` is connected to the trigger port of the `band pass filter1` Simulink block. The `band pass filter1` block has its **Trigger type** field set to **Function Call**.



This sequence is followed when state A is active and the transition from state A to state B is valid and is taken:

1. State A exit actions execute and complete.
2. State A is marked inactive.
3. The transition action is executed and completed.

   In this case the transition action is a broadcast of `event1`. Because `event1` is an event output to Simulink with a function call trigger, the `band pass filter1` block executes and completes, and then returns to the next statement in the execution sequence. The value of `y` is fed back to the Stateflow diagram.
4. State B is marked active.
5. State B entry actions execute and complete (`x = x + y`). The value of `y` is the updated value from the `band pass filter1` block.
6. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

Defining Edge-Triggered Output Events

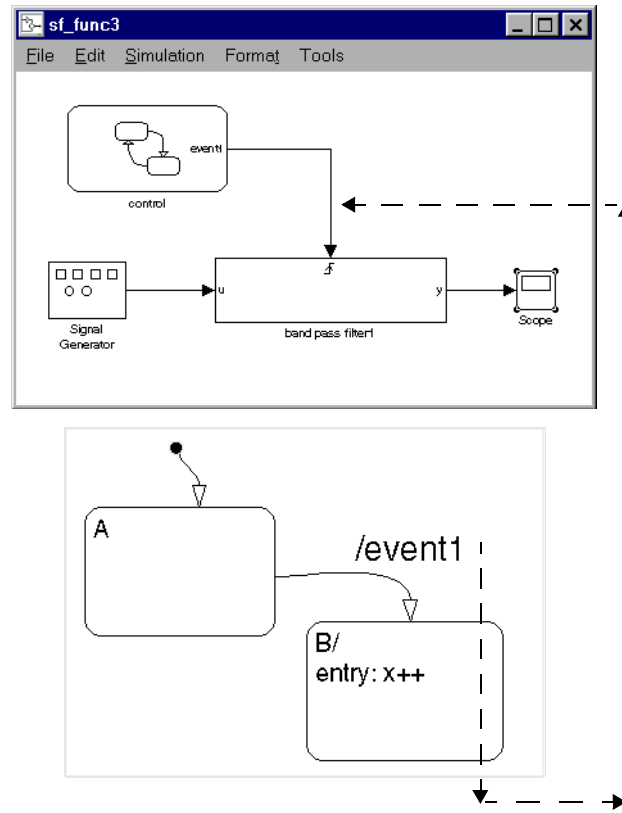
Simulink controls the execution of edge-triggered subsystems with output events. These are essential conditions that define this use of triggered output events:

- The chart has an **Output to Simulink** event with the trigger type **Either Edge**. See "Defining Output Events" on page 6-9.
- The Simulink block connected to the edge-triggered **Output to Simulink** event has its own trigger type set to the equivalent edge triggering.

Edge-Triggered Semantics Example

In this example the transition from state A to state B (in the Stateflow diagram) has a transition action that specifies the broadcast of `event1`. `event1` is defined in Stateflow to be an **Output to Simulink** with an **Either edge** trigger type. The Stateflow block output port for `event1` is connected to the trigger port of

the `band pass filter1` Simulink block. The `band pass filter1` block has its **Trigger type** field set to **Either edge**.



This sequence is followed when state `A` is active and the transition from state `A` to state `B` is valid and is taken:

1. State `A` exit actions execute and complete.
2. State `A` is marked inactive.
3. The transition action, an edge-triggered **Output to Simulink** event broadcast, is registered (but not executed). Simulink is controlling the execution and execution control does not shift until the Stateflow block completes.

4. State `B` is marked active.

5. State `B` entry actions execute and complete (`x = x++`).

6. The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

7. The `band pass filter1` block is triggered, executes, and completes.

MATLAB Workspace Interfaces

The MATLAB workspace is an area of memory normally accessible from the MATLAB command line. It maintains a set of variables built up during a MATLAB session.

Examining the MATLAB Workspace in MATLAB

Two commands, `who` and `whos`, show the current contents of the workspace. The `who` command gives a short list, while `whos` also gives size and storage information.

To delete all the existing variables from the workspace, enter `clear` at the MATLAB command line.

See the MATLAB online or printed documentation for more information.

Interfacing the MATLAB Workspace in Stateflow

Stateflow charts have the following access to the MATLAB workspace:

- You can access MATLAB data or MATLAB functions in Stateflow action language with the `ml` namespace operator or the `ml` function.

  See “MATLAB Functions and Data” on page 7-46 for more information.
- You can use the MATLAB workspace to initialize chart data at the beginning of a simulation.

  See the subsection “Initialize from” on page 6-22 of the topic “Setting Data Properties” on page 6-16.
- You can save chart data to the workspace at the end of a simulation.

  See “Save final value to base workspace” on page 6-24 for more information.

Interface to External Sources

Any source of data, events, or code that is outside a Stateflow diagram, its Stateflow machine, or its Simulink model, is considered external to that Stateflow diagram. Stateflow can interface data and events from external sources to your Stateflow chart, as described in the following topics:

- "Exported Events" on page 8-23 — Describes events that a Stateflow chart exports to locations outside itself.
- "Imported Events" on page 8-25 — Describes events that a Stateflow chart imports from locations outside itself.
- "Exported Data" on page 8-27 — Describes data that a Stateflow chart exports to locations outside itself.
- "Imported Data" on page 8-29 — Describes data that a Stateflow chart imports from locations outside itself.

See Chapter 6, "Defining Events and Data," for information on defining events and data.

You can include external source code in the **Target Options** section of the **Target Builder** dialog box. (See "Building Custom Code into the Target" on page 11-4.)

Exported Events

Consider a real-world example to clarify when to define an **Exported** event. You have purchased a communications pager. There are a few people you want to be able to page you, so you give those people your personal pager number. These people now know your pager number and can call that number and page you at any time. You do not usually page yourself, but you can do so. Telling someone the pager number does not mean they have heard and recorded the number. It is the other person's responsibility to retain the number.

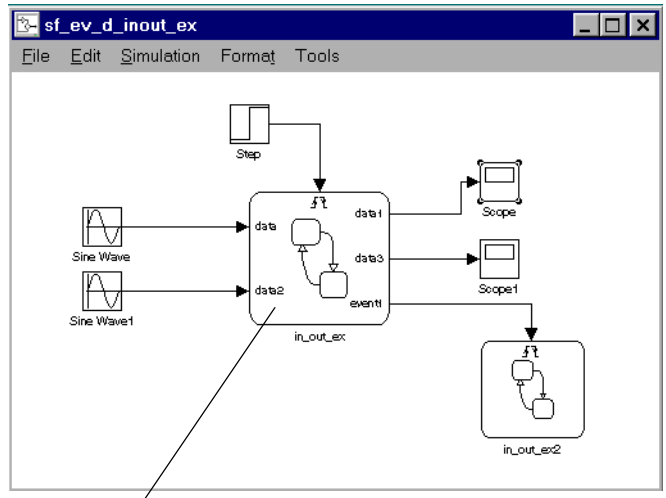
Similarly, you might want an external source (outside the Stateflow diagram, its Stateflow machine, and its Simulink model) to be able to broadcast an event. By defining an event's scope to be **Exported**, you make that event available to external sources for broadcast purposes. Exported events must be parented by the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. The Stateflow machine also retains the ability to broadcast the exported event. Exporting the

event does not imply anything about what the external source does with the information. It is the responsibility of the external source to include the **Exported** event (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source for the event is another Stateflow machine, then that machine must define the event as an **Exported** event and the other machine must define the same event as **Imported**. Stateflow generates the appropriate export and import event code for both machines.

Exported Event Example

This example shows the format required in the external code source (custom code) to take advantage of an **Exported** event generated in Stateflow.



e is added and defined as an **Exported** event.

Stateflow generates the following code for the **Exported** event:

```
void broadcast_e (void)
{
/* code based on
   event definition
*/
 } ...
```

e is imported in the external code source.

External code source

```
void func_example(void)
{
extern void broadcast_e (void);
 ...
external_broadcast_e();
...
 }
```

Imported Events

The preceding pager example for exported events can clarify the use of imported events. For example, someone buys a pager and tells you that you might want to use this number to page them in the future and they give you the pager number to record. You can then use that number to page that person.

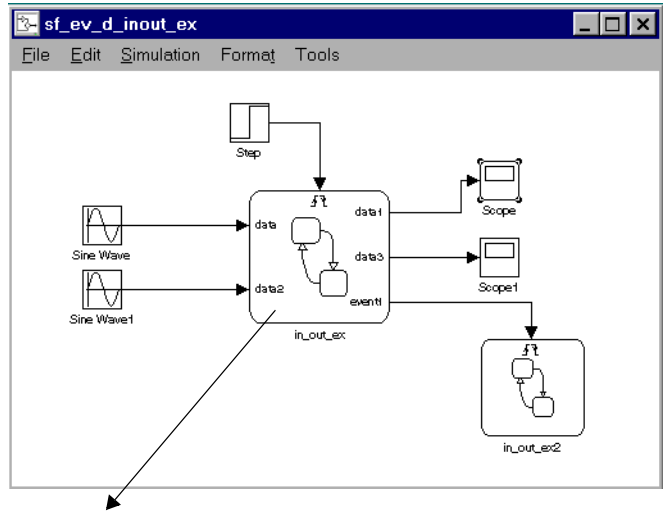
Similarly, you might want to broadcast an event that is defined externally (outside the Stateflow diagram, its Stateflow machine, and its Simulink model). By defining an event's scope to be **Imported**, you can broadcast the event anywhere within the hierarchy of that machine (including any offspring of the machine).

An imported event's parent is external. However, the event needs an adoptive parent to resolve symbols for code generation. An imported event's adoptive parent must be the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. It is the responsibility of the external source to make the imported event available (in the manner appropriate to the source).

If the external source is another Stateflow machine, the source machine must define the same event as **Exported**. Stateflow generates the appropriate import and export event code for both machines.

Imported Event Example

The following example shows the format required in an external code source (custom code) to generate an **Imported** event in Stateflow.



e is added and defined as an **Imported event.**

Stateflow generates the following code for the **Imported** event:

```
extern void broadcast_e (void);
```

e is exported in the external code source.

External code source

```
void broadcast_e (void)
{
...
}
```

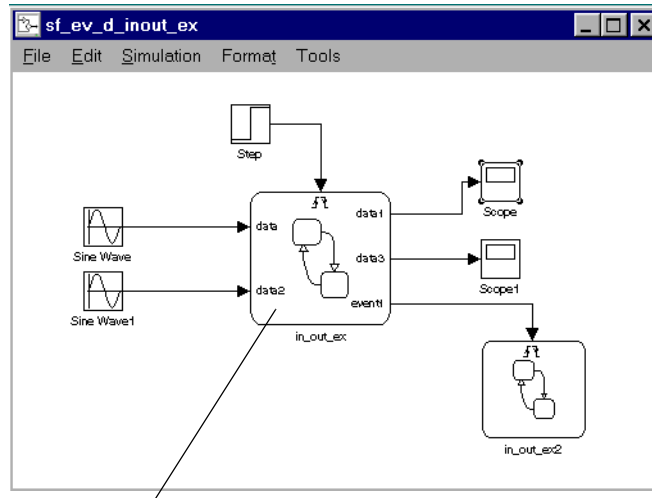
Exported Data

You might want an external source (outside the Stateflow diagram, its Stateflow machine, and itse Simulink model) to be able to access a data object. By defining a data object's scope as **Exported**, you make it accessible to external sources. Exported data must be parented by the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. The Stateflow machine also retains the ability to access the exported data object. Exporting the data object does not imply anything about what the external source does with the data. It is the responsibility of the external source to include the exported data object (in the manner appropriate to the source) to make use of the right to access the data.

If the external source is another Stateflow machine, then that machine defines an exported data object and the other machine defines the same data object as imported. Stateflow generates the appropriate export and import data code for both machines.

Exported Data Example

The following example shows the format required in the external code source (custom code) to import a Stateflow exported data object:



ext_data is added and defined as an **Exported** data.

Stateflow generates the following code for the **Exported** data:

```
int ext_data;
```

ext_data is defined as imported in the external code source.

External code source

```
extern int ext_data;

void func_example(void)
{
...

ext_data = 123;
...
 }
```

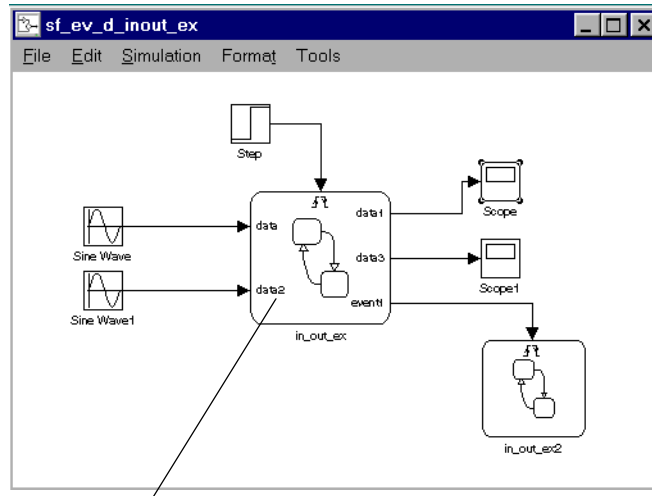
Imported Data

Similarly, you might want to access a data object that is externally defined outside the Stateflow diagram, its Stateflow machine, and its Simulink model. If you define the data's scope as **Imported**, the data can be accessed anywhere within the hierarchy of the Stateflow machine (including any offspring of the machine). An imported data object's parent is external. However, the data object needs an adoptive parent to resolve symbols for code generation. An imported data object's adoptive parent must be the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. It is the responsibility of the external source to make the imported data object available (in the manner appropriate to the source).

If the external source for the data is another Stateflow machine, that machine must define the same data object as **Exported**. Stateflow generates the appropriate import and export data code for both machines.

Imported Data Example

This example shows the format required to retrieve imported data from an external code source (custom code).



ext_data is defined as exported in the external code source.

ext_data is added and defined as an imported data Stateflow generates the following code for the imported data:

```
extern int ext_data;
```

External code source

```
int ext_data;

void func_example(void)
{
...
}
```

9

Truth Tables

You use truth tables to instruct Stateflow to provide logical behavior that you specify with conditions and outcomes. Users can always create any function they want in Stateflow with graphical functions using flow graph notation. However, truth tables give you the convenience of specifying functions with the logical behavior you specify without having to draw flow graphs. This saves you the time of constructing your own functions, and helps you simplify and condense lengthy action language statements in your diagrams into a single function call. To learn how to construct successful truth tables, see the following sections:

Introduction to Truth Tables (p. 9-2)	Gives an overview to the concept of a truth table and the process of creating, specifying, generating, and using a truth table in Stateflow. It also includes a simple “your first truth table” example.
Using Truth Tables (p. 9-14)	Shows you why and how truth tables are used in Stateflow diagrams.
Specifying Stateflow Truth Tables (p. 9-18)	Describes all aspects of creating a new truth table and filling it out.
Edit Operations in a Truth Table (p. 9-37)	A complete reference on the editing operations available to you in the truth table editor.
Error Checking for Truth Tables (p. 9-48)	Provides you with a reference of errors and warnings that Stateflow detects for truth tables at the start of simulation.
Debugging Truth Tables (p. 9-56)	Shows you how you can debug a truth table by simulated execution of individual conditions and decisions.
Model Coverage for Truth Tables (p. 9-64)	Displays an example model coverage report for a truth table and interprets the results.
How Stateflow Realizes Truth Tables (p. 9-68)	Shows you the graphical function that Stateflow generates to realize a truth table.

Introduction to Truth Tables

Stateflow truth tables implement functions with the logical behavior that you specify for them with conditions, decisions, and actions. The following sections introduce you to the general concept of a truth table and give you an overview of how you use them in Stateflow that includes an example model using a truth table for you to build.

- "What Are Truth Tables?" on page 9-2 — Describes the truth table concept that Stateflow realizes.
- "Using Truth Tables in Stateflow" on page 9-4 — Gives you a start to finish overview of using truth tables in Stateflow.
- "Your First Truth Table" on page 9-7 — Introduces you to truth tables with a simple but very concrete example.

After your introduction to truth tables, you might want to continue by looking at a more complete description of the process for using truth tables in "Specifying Stateflow Truth Tables" on page 9-18.

What Are Truth Tables?

Truth tables are a general concept that Stateflow uses to realize the logical behavior that you specify as a function that you can call in action language. Stateflow truth tables contain a table of conditions, condition outcomes, decision outcomes, and actions arranged with a structure similar to the following:

Truth Table Structure

Condition	**Decision 1**	**Decision 2**	**Decision 3**	**Decision 4**
x == 10	T	F	F	-
x == 20	F	T	F	-
x == 30	F	F	T	-
Action	y = 10	y = 20	y = 30	y = 40

Stateflow evaluates this truth table with the following rules:

- A condition entered in the truth table **Condition** column must evaluate to a value of true (nonzero) or false (zero).
- Each condition is assumed to be independent of every other condition.
- Possible values for each condition can be T (true), F (false), or - (true or false). These are referred to as condition outcomes.
- A decision outcome is a column of condition outcomes linked together with a logical AND.

  For example, you can express the Decision 1 column in pseudocode as the following:

  ```
  if ((x == 10) & !(x == 20) & !(x == 30))
  ```

- Each decision outcome column has a single action specified for it at the bottom of its column.

  For example, the action specified for Decision 1 in the preceding example is `y = 10`. This means that Decision 1 and its action can be expressed with the following pseudocode:

  ```
  if ((x == 10) & !(x == 20) & !(x == 30))
     y = 10;
  ```

- Decision outcomes are tested in left to right column order.
- If a decision outcome is tested as true, its assigned action is immediately executed and the truth table is exited.
- The order of testing for individual condition outcomes for a decision outcome is undefined.

  The current implementation of truth tables in Stateflow evaluates the conditions for each decision in top-down order. Because this implementation is subject to change in the future, users must not depend on a particular evaluation order.

- You can enter a default decision outcome covering all possible decision outcomes as the last (rightmost) decision outcome in the table.

  The default decision outcome covers any remaining decision outcomes not tested for in the preceding decision outcome columns.

Using the preceding rules for Stateflow truth tables, the preceding example truth table is realized with the following pseudocode:

```
if ((x == 10) & !(x == 20) & !(x == 30))
   y = 10;
```

```
elseif (!(x == 10) & (x == 20) & !(x == 30))
   y = 20;
elseif (!(x == 10) & !(x == 20) & (x == 30))
   y = 30;
else
   y = 40;
endif;
```

See "Using Truth Tables in Stateflow" on page 9-4 to see how Stateflow has adapted this general concept to realize Stateflow truth tables. Also, if you're interested in testing the preceding truth table, see the topic "Your First Truth Table" on page 9-7.

Initial and Final Actions

With regard to truth table execution in Stateflow, the preceding rules are complete with the exception of the initial and final actions that Stateflow provides.

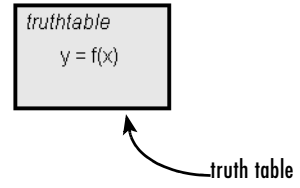
You can specify an initial action for a truth table that executes before any decision outcomes are tested. You can also specify a final action that executes as the last action before the truth table is exited. Entering these special actions is described in detail in "Special INIT and FINAL Actions" on page 9-29.

Using Truth Tables in Stateflow

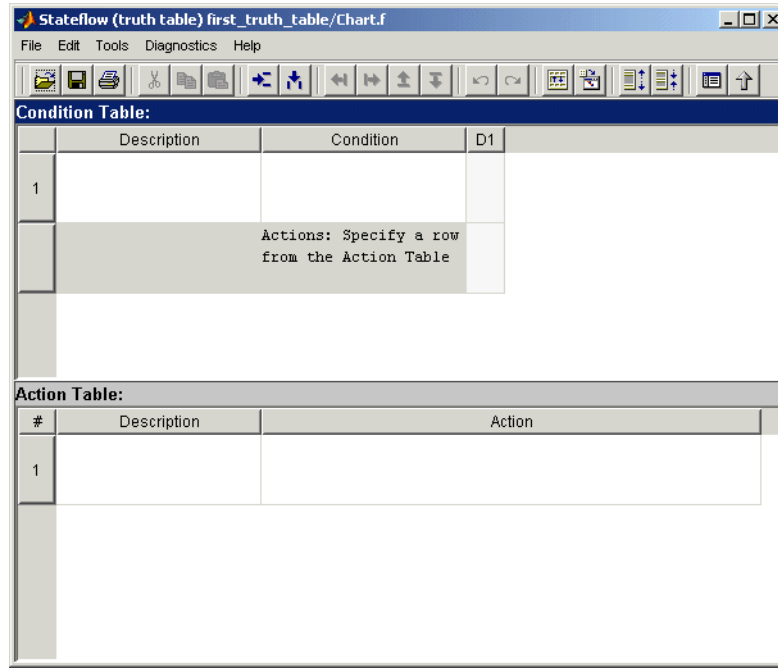
You are introduced to the concept of a truth table in "What Are Truth Tables?" on page 9-2. This topic gives you an overview of the following process for creating and using a truth table in Stateflow:

1 Use the truth table drawing tool in the Stateflow diagram editor to draw a new truth table (see "How to Create a Truth Table" on page 9-18).

  The following example shows a truth table in a simple Stateflow diagram:

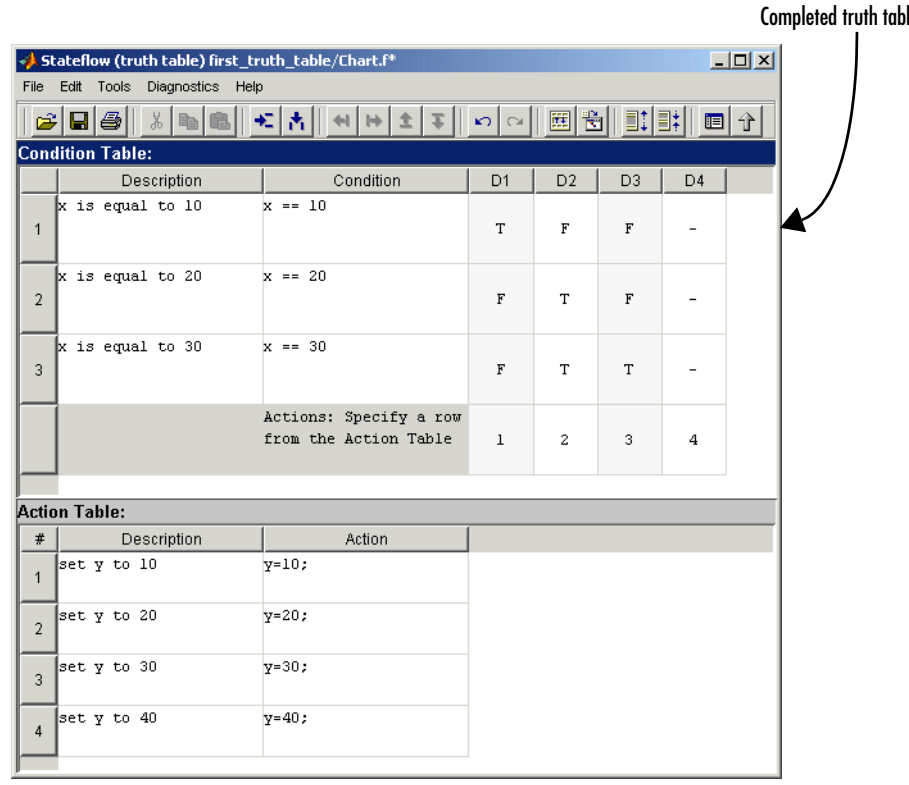


2 Double-click the truth table in the Stateflow editor to enter the truth table editor, which appears as follows:



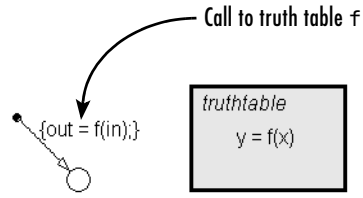
3 Use the truth table editor to specify the conditions, condition outcomes, decision outcomes, and actions that define the logical behavior you want for it.

A finished truth table has an appearance similar to the following:



4 Call the truth table function in Stateflow action language.

In the following Stateflow chart example, the default transition to a junction calls the truth table to evaluate the data `out` when the chart is executed:



Later, when you decide to simulate your model, Stateflow tests your completed truth tables for missing or undefined information and possible logical conflicts. See “Error Checking for Truth Tables” on page 9-48.

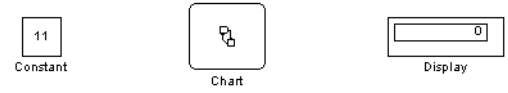
Stateflow realizes truth table functions as a flow diagram in a graphical function that it automatically generates at the start of simulation. See “How Stateflow Realizes Truth Tables” on page 9-68 for a description of the structure of a graphical function for a truth table.

Now that you have seen an overview of the process for creating truth tables in Stateflow, it’s time for you to experience it yourself in “Your First Truth Table” on page 9-7.

Your First Truth Table

In this topic, you create the truth table in “What Are Truth Tables?” on page 9-2. To execute the truth table, you also need a Stateflow chart that calls the truth table and a Simulink model that calls the Stateflow chart. The following steps guide you through this process:

1 Create the following simple model in Simulink.

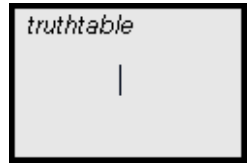


In the **Simulation Parameters** dialog, leave the **Solver Options** field set at **Variable-step** but change the **Stop Time** to inf.

2 Double-click the Stateflow block (named Chart in the preceding example) to open the empty Stateflow diagram.

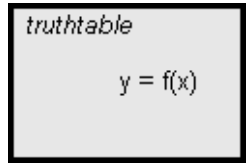
3 Select the Truth Table button from the Stateflow diagram editor drawing toolbar .

4 Move your cursor into the empty diagram area and click to place a new truth table.

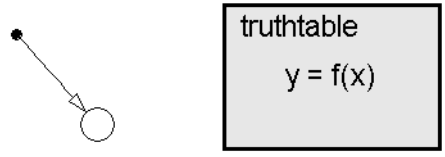


The new truth table is highlighted and has a flashing text cursor in a text field in the middle of the truth table.

5 Type `y = f(x)` in the text field, then press the Esc key when finished.

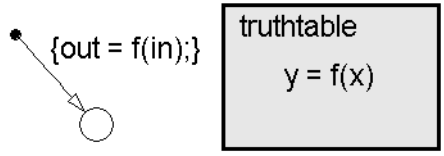


6 Draw a junction with a default transition entering it to the left of the truth table.



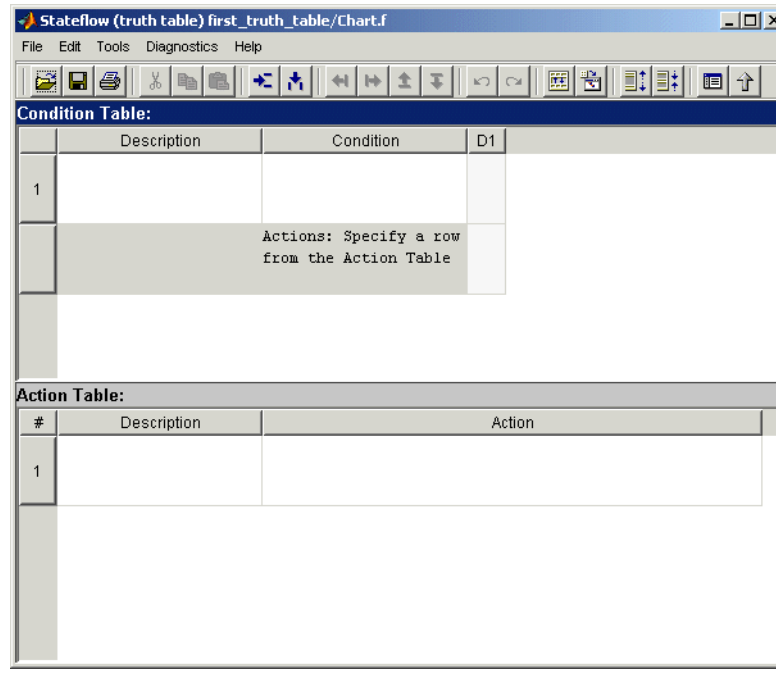
7 Label the default transition with the text `{out = f(in);}`.

This assigns a condition action to the default transition that calls the truth table. Your finished Stateflow diagram should have the following appearance:



8 Double-click the truth table to specify its contents.

The empty truth table appears as follows:



Notice that the header of the **Condition Table** is already highlighted. If it is not, click anywhere in the **Condition Table** to select it.

9 Perform the following edit operations to add rows and columns to the **Condition Table**:

a Click the Append Row toolbar button twice to add two rows to the **Condition Table**.

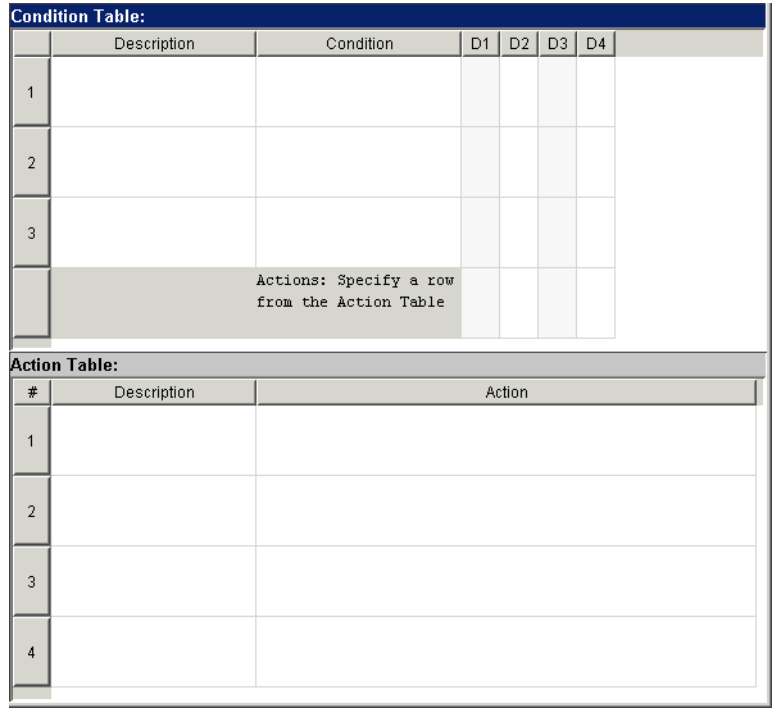
b Click the Append Column toolbar button three times to add three columns to the **Condition Table**.

10 Perform the following edit operations to add rows to the **Action Table**:

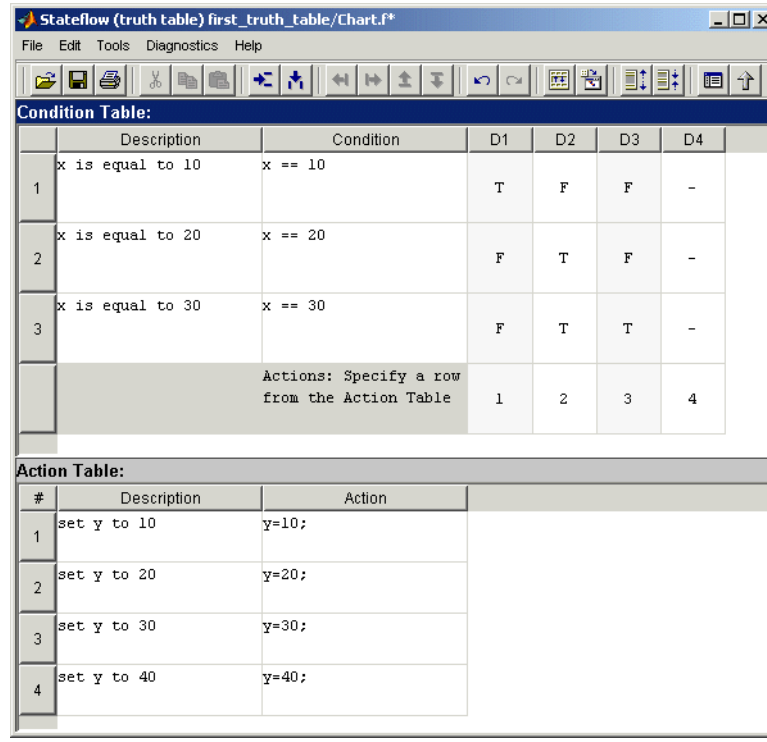
a Click anywhere in the **Action Table** to select it.

b Click the Append Row toolbar button three times to add three rows to the **Action Table**.

11 Click and drag the borders of the truth table editor window and the separator between the **Condition Table** and the **Action Table** to enlarge it to show all rows and columns of the full tables.



12 Edit the individual cells of the table with the entries displayed in the following finished truth table:



You can use some of the following edit operations to accomplish this:

- Click an individual cell to edit its contents.
- Use the Tab key to move to the next horizontal cell on the right. Use Shift-Tab to move left.
- For the cells with a T, F, or - entry, use the up and down arrow keys to move to an adjacent cell in the vertical direction. Enter an upper- or lowercase T or F or the - character or press the space bar to toggle among these values.

  See “Edit Operations in a Truth Table” on page 9-37 for full details, especially if you find that you need to insert or delete a row or column.

13 Specify the data `in` for the Stateflow diagram with the following steps:

a Select the Goto Explorer toolbar button to enter Stateflow Explorer.

b In the resulting Explorer window, select the Chart level in the **Object Hierarchy** pane on the left.

c From the **Add** menu select **Data**.

d In the **Contents** pane on the right side, double-click **data** in the **Name** column.

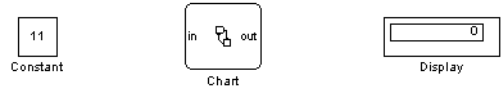
e In the resulting text field, change the name to `in` and press Return.

f Click **Local** under the **Scope** column.

g In the resulting menu, select **Input from Simulink**.

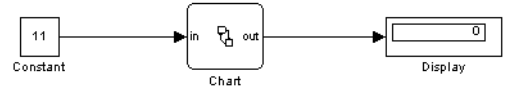
14 Use the procedure from the preceding step to add the data `out` with a scope of **Output to Simulink**.

Notice that ports for the Stateflow data `in` and `out` now appear on the Stateflow block in Simulink.



In the preceding steps you used the Stateflow Explorer tool to add data to Stateflow that you use in the Stateflow diagram. This is a powerful tool for adding the data and events that you need in your Stateflow diagrams. See “The Stateflow Explorer Tool” on page 10-3 for full details on how to use the Stateflow Explorer.

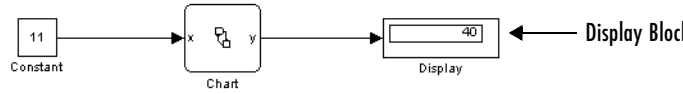
15 Complete connections on the Simulink diagram as follows:



16 Press the Start Simulation button in either the Simulink or Stateflow diagram editor windows to begin simulating the model you've created.

If you get any errors or warnings, refer to "Error Checking for Truth Tables" on page 9-48.

In Simulink, the Display block should display as follows.



17 Continue testing by changing the value of the Constant block during simulation.

Do this by double-clicking on the Constant block and entering a new value in the **Constant value** field for the resulting Block Parameters dialog.

Now that you have an idea of what it's like to create a truth table, continue by taking a more detailed look at the process in "Specifying Stateflow Truth Tables" on page 9-18.

Using Truth Tables

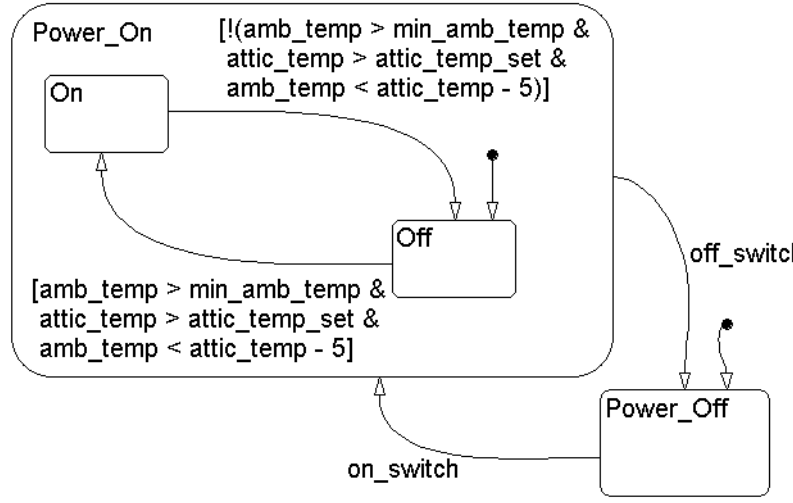
The previous section, "Introduction to Truth Tables" on page 9-2, introduces you to truth tables. This section shows you how using truth tables in Stateflow diagrams gives you both convenience and power for specifying logical behavior for your diagrams in the following topics:

- "Why Use a Truth Table?" on page 9-14 — Shows you why truth tables can clear up diagrams and save you the time of writing flow graphs to implement logical behavior.
- "Calling Truth Table Functions in Stateflow" on page 9-16 — Gives you the rules for when and how you call truth tables in Stateflow.

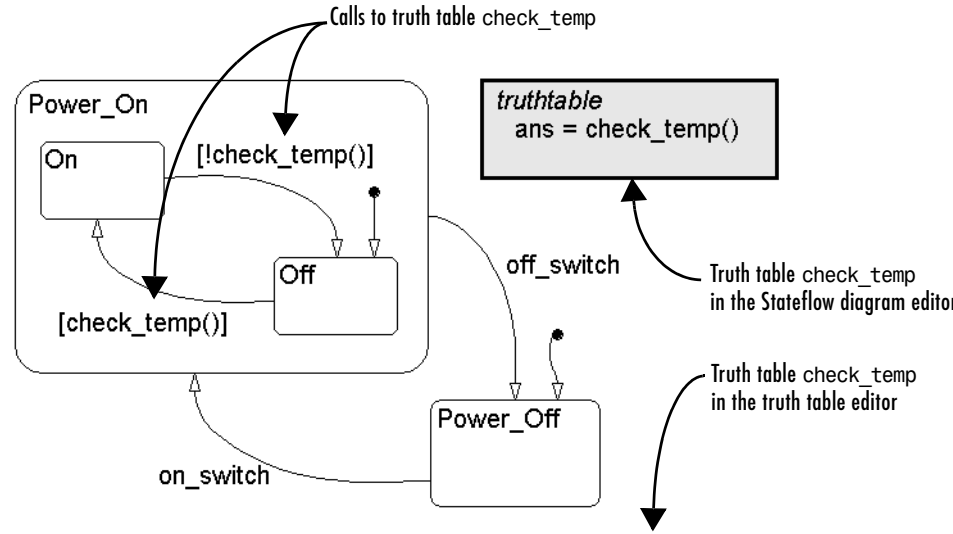
Continue with "Specifying Stateflow Truth Tables" on page 9-18 for a complete description on implementing truth tables in Stateflow.

Why Use a Truth Table?

Every truth table begins as a need to create a function that you can call in the action language of your Stateflow diagrams. The following example shows conditions that guard the transitions between the states Power_Off and Power_On.



These conditions check the ambient air temperature, the attic temperature, and the attic temperature setting to see if it's time to turn an attic fan on or off. Because so many values are checked, these conditions are long and clog up your diagram, which forces it to be bigger and less readable than you would like. Instead, replace all these conditions by a call to a truth table that conveniently encapsulates their logical behavior as follows:



Condition Table:

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-
		Actions: Specify a row from the Action Table	ON	OFF

Action Table:

#	Description	Action
1	Stay on or turn on	ON: ans = 1;
2	Stay off or turn off	OFF: ans = 0;

In this example, the return value from the call to `check_temp` is placed in conditions that guard transitions between the On and Off states. During simulation, if `check_temp` returns a positive value, the transition from state Off to state On can take place if the Off state is active. If it returns a value of 0, the transition from On to Off can take place if the On state is active.

Although you can accomplish the same behavior with a graphical function, with a truth table function you can use simpler textual programming logic instead of writing a Stateflow flow diagram.

Now that you know the value of truth tables, continue by learning the rules for when and how to call truth tables, in the topic "Calling Truth Table Functions in Stateflow" on page 9-16.

Calling Truth Table Functions in Stateflow

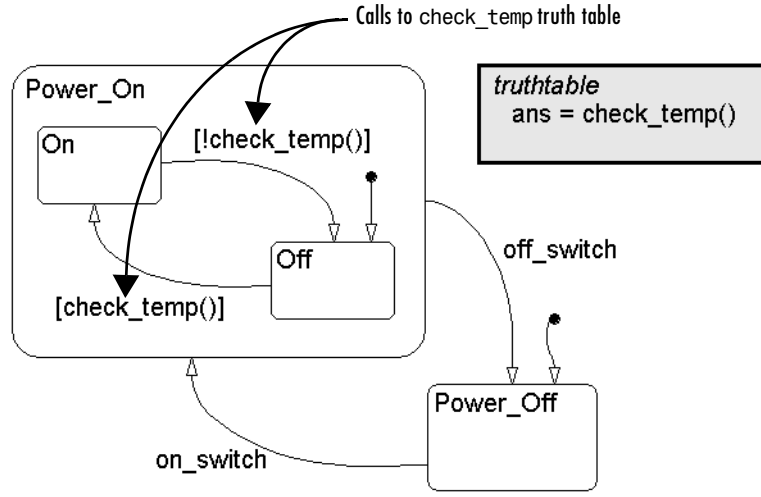
The previous topic, "Why Use a Truth Table?" on page 9-14, answers the why question for truth tables. This topic answers the question of where to use truth tables. Later, the section "Specifying Stateflow Truth Tables" on page 9-18 addresses the lengthier question of how.

Because Stateflow realizes truth tables internally with a generated graphical function (see "How Stateflow Realizes Truth Tables" on page 9-68), the rules for calling truth table functions coincide with those for calling graphical functions and include the following:

- You call a truth table by using its signature in the label of the truth table box in the Stateflow diagram editor as shown:

  ```
  return_value = function_name (argument1, argument2, ...)
  ```

  The following example calls a truth table function in the conditions of two transitions:



In this example, the return value from the call to `check_temp` is placed in conditions that guard transitions between the states On and Off. During simulation, if `check_temp` returns a positive value and the Off state is active, the transition from state Off to On takes place. If `check_temp` returns a value of 0 and the state On is active, the transition from state On to state Off takes place.

- You can call truth table functions from other truth tables, including recursive calls to the same truth table.
- You can call truth table functions from graphical functions (and you can call graphical functions from the conditions and actions of truth table functions).
- You can call exported truth tables from any Stateflow chart in the model.

  This means that other charts in the model can call the exported truth table function of a particular Stateflow chart. See “Exporting Graphical Functions” on page 5-53.

Specifying Stateflow Truth Tables

The beginning section, "Introduction to Truth Tables" on page 9-2, introduces you by example to the concept of a truth table and an overview of its use in Stateflow. The following section, "Using Truth Tables" on page 9-14, shows you the convenience and power of using truth tables in Stateflow diagrams. This section takes you through the entire process of creating, specifying, and calling truth tables with the following topics:

- "How to Create a Truth Table" on page 9-18 — Gives you the procedure for creating and labeling an empty truth table function in a Stateflow diagram.
- "Editing a Truth Table" on page 9-20 — Gives you an overview procedure for filling out the different parts of a truth table.
- "Entering Conditions" on page 9-23 — Shows you how to enter conditions in a truth table.
- "Entering Decision Outcomes" on page 9-25 — Shows you how to enter decision outcomes for the conditions you enter in a truth table.
- "Entering Actions for Decision Outcomes" on page 9-26 — Shows you how to enter actions for each decision outcome you enter in a truth table.
- "Using Stateflow Data and Events in Truth Tables" on page 9-32 — Describes the access that a truth table has to the data and events defined for Stateflow chart objects.
- "Specifying Properties for a Truth Table" on page 9-34 — Shows you how to specify the properties for a truth table and describes each property.

Before you begin this extensive section on specifying truth tables, you might want to visit the next section, "Edit Operations in a Truth Table" on page 9-37, to familiarize yourself with the edit operations used for specifying truth tables in the truth table editor.

Later, when you start constructing your own truth tables, you will need to troubleshoot them with the aid of truth table error checking. See "Error Checking for Truth Tables" on page 9-48 for details on the errors an warnings you receive.

How to Create a Truth Table

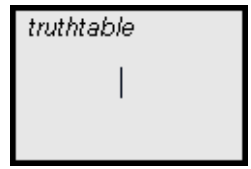
You first create an empty truth table in the Stateflow diagram editor before you open it to specify its logical behavior (see "Editing a Truth Table" on page 9-20).

Do the following to create an empty (unspecified) truth table in a Stateflow diagram:

1. Select the Truth Table button in the diagram editor drawing toolbar and move the cursor to an empty area of the diagram editor space.

   Notice that the mouse cursor takes on the shape of a box with oval corners.

2. Click an appropriate location to place the new truth table in the diagram editor.

   The truth table now appears as a box with sharp corners whose contents are shaded. The box is titled **truthtable**. A text cursor appears in the middle of the truth table for entering its signature.

3. Type the signature for the truth table function.

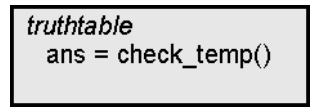
   In the signature for a truth table you specify arguments and a return value for the truth table function as in the following example:

   ```
   ans = check_temp()
   ```

   If you omit the parentheses `()` for a signature that has no arguments, Stateflow provides them automatically. Otherwise, type the full signature.

4. Press the Esc key when you are finished entering the text for the signature.

Now that you have labeled your truth table, it should have the following appearance:



If you need to reedit the truth table function signature,

1 Click the function signature text.

  This places an editing cursor in the text of the signature.

2 Edit the signature and press the Esc key when finished.

You can also set the signature of your truth table function any time by changing the **Label** property in the **Properties** dialog for the truth table. See "Specifying Properties for a Truth Table" on page 9-34

Note You can use the return and argument variables you specify in the signature when you specify conditions and actions for a truth table. See "Using Stateflow Data and Events in Truth Tables" on page 9-32 for a description of the data you can use to specify conditions and actions in truth tables.

Like any object in a Stateflow diagram, you can copy and paste a truth table in the Stateflow diagram editor from one location to another. See "Copying Objects" on page 5-14 for a detailed procedure.

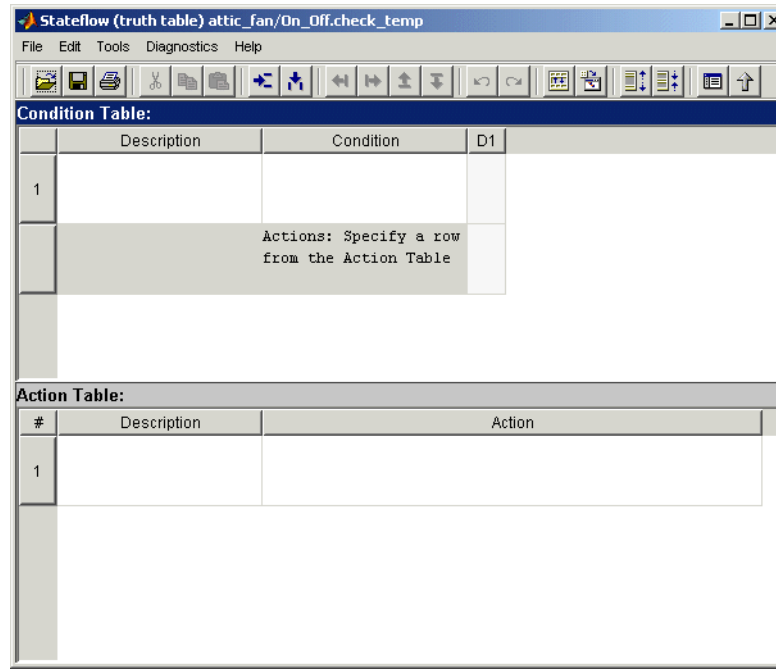
Editing a Truth Table

The truth table that you create and label in a Stateflow diagram (see "How to Create a Truth Table" on page 9-18) appears as a shaded box with sharp corners whose title is `truthtable`. Now you must specify its logical behavior by editing its contents in the truth table editor.

Use the following steps to begin editing the `check_temp` truth table:

1 Access the truth table editor by double-clicking the truth table in the Stateflow diagram editor.

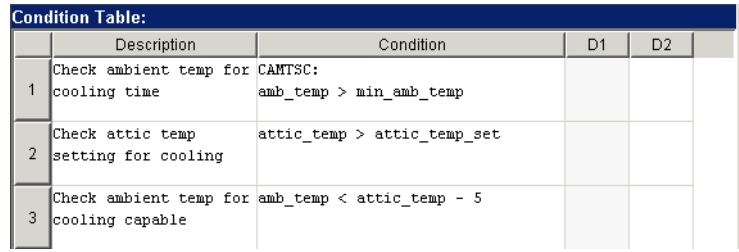
  A blank truth table editor window appears.



The truth table editor has two tables: the **Condition Table** and the **Action Table**. You edit these tables to specify the logical behavior of your truth table. Before you begin specifying the contents of the truth table, you want to add rows and columns to each table as you did in "Your First Truth Table" on page 9-7. For these and other basic editing operations, see "Edit Operations in a Truth Table" on page 9-37.

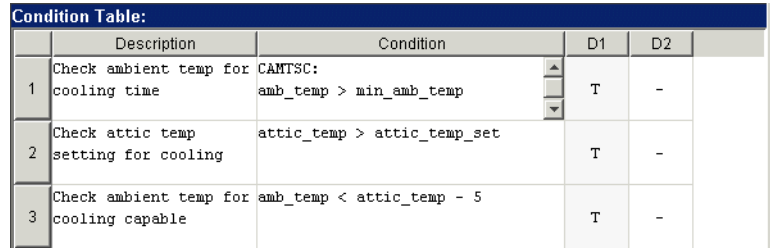
2 Enter condition descriptions (optional) and conditions in each row of the **Condition Table**, except for the last row, the **Actions** row.

See the topic "Entering Conditions" on page 9-23 for complete details. The following shows the filled in conditions for the example `check_temp` truth table used in this topic:



3 Enter the following decision outcomes in the columns labeled **D1**, **D2**, and so on.

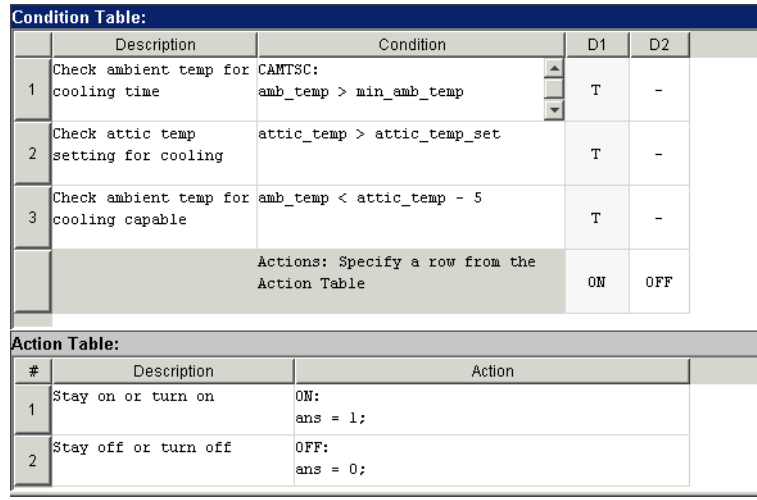
See the topic "Entering Decision Outcomes" on page 9-25 for a detailed description of this process. The following shows the filled in decision outcomes for the example truth table `check_temp` used in this topic:



A decision outcome column combines outcomes for each condition into a decision outcome. The allowed values for each condition outcome are T (true), F (false), or - (true or false).

4 Specify the actions for each decision outcome.

The following shows a filled in **Actions** row and **Action Table** for the example truth table `check_temp` used in this topic:



You specify actions for decision outcomes as follows:

a For each decision outcome column, specify a label or row number in the **Actions** row of the **Condition Table**.

This label or row number maps to an action with the same label or row number in the **Action Table**. You can also specify multiple actions with multiple row numbers or labels.

b For actions that you assign to decision outcomes with row numbers, enter those actions in the corresponding rows of the **Action Table**.

c For actions that you assign to decision outcomes with labels, enter those actions with the appropriate label in any rows of the **Action Table**.

See the topic “Entering Actions for Decision Outcomes” on page 9-26 for a more detailed description of this process.

Entering Conditions

You edit a truth table to specify its logical behavior in the form of conditions, decisions, and actions (see “Editing a Truth Table” on page 9-20). Enter conditions in the first two columns of the rows of the **Condition Table** as follows:

1 In the **Description** column, click a cell and enter an optional description of this condition.

   Condition descriptions are carried into the action language for evaluation of this condition in the generated diagram and into the resulting generated code as code comments. See "How Stateflow Realizes Truth Tables" on page 9-68.

2 Press the Tab key to advance to the corresponding cell in the **Condition** column, and enter the following:

   a Enter an optional symbolic label for this condition followed by a colon (`:`).

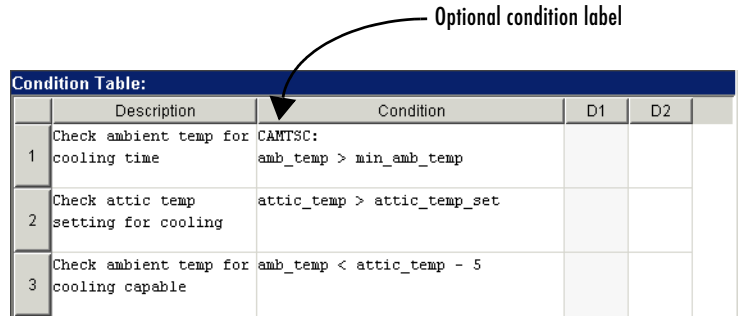
   Stateflow uses this symbol for the name of a temporary data variable to store the outcome of this condition in generated code. If no label is entered, Stateflow names the variable itself.

   Condition labels must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (_).

   b After the optional label, enter the condition tested for in the generated diagram.

   Enter conditions as you would in Stateflow action language, with or without optional bracket characters. See "Conditions" on page 2-16.

The following example shows conditions entered for the `check_temp` truth table function to control an attic exhaust fan:



The conditions in this example do the following:

- Check the ambient air temperature to see if it is cooling season
- Check if the attic is hot enough to need cooling
- Check the ambient air temperature to see if the fan can cool the attic

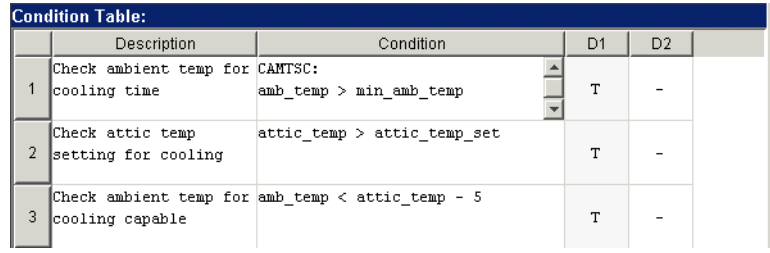
You can use data defined for the Stateflow chart of the truth table to specify conditions. You can also use data passed to the truth table function through the arguments and return value of its signature. See "Using Stateflow Data and Events in Truth Tables" on page 9-32 for more detail on using Stateflow data and events in truth table conditions and actions.

Entering Decision Outcomes

After you enter conditions in the truth table editor ("Entering Conditions" on page 9-23), enter the expected outcomes of each condition in the columns labeled **D1**, **D2**, and so on.

Each decision outcome column binds a group of condition outcomes together with an AND relationship (see "What Are Truth Tables?" on page 9-2). When that decision outcome is realized during execution, the action specified in the **Action Table** for that column is executed (see "Entering Actions for Decision Outcomes" on page 9-26).

The following shows the decision outcomes entered for the `check_temp` truth table in columns **D1** and **D2**:



Use the following procedure to enter decision outcome values for columns **D1** and **D2** of the `check_temp` truth table:

1 Click a particular cell in a decision column.

2 Specify an expected outcome by entering a value of T, F, or -, or press the space bar to toggle through these available values.

3 Use the arrow keys to access the next decision outcome cell.

For the `check_temp` truth table, only the decision outcome in which all three conditions are true is tested for. All other possible decision outcomes are relegated to the last column, a default decision outcome.

Specifying Default Decision Outcomes

Normally, you want to specify only a few of all the possible decision outcomes for a generated function. In the previous example, with three conditions specified, there can be as many as $2^3 = 8$ possible decision outcomes. However, usually only a few decision outcomes are important enough to test for. In the truth table function `check_temp`, only the decision outcome in which all conditions are true turns the attic fan on. All other outcomes result in the attic fan remaining off or being turned off.

To indicate a decision outcome for all decision outcomes but those tested for, enter a last decision outcome column with an entry of - (true or false) for all conditions. This is the default decision outcome column. You can also specify an action for the default decision outcome column.

Note The default decision outcome column must be the rightmost column in the **Condition Table**.

Entering Actions for Decision Outcomes

After you have entered decision outcomes for the conditions in the truth table ("Entering Decision Outcomes" on page 9-25), enter and assign actions to each decision outcome as follows:

1 Click a cell for a decision outcome in the **Actions** row of the **Condition Table** and enter a label or row number action specifier for an action in the **Action Table**.

   You can specify multiple actions for a decision outcome by separating action specifiers with a comma.

2 Click the cell in the **Description** column of the appropriate row (see note below) of the **Action Table** and enter an optional description of the action.

   Action descriptions are carried into the action language for evaluation of this action in the generated diagram and into the resulting generated code as code comments. See "How Stateflow Realizes Truth Tables" on page 9-68.

3 Press the Tab key to move to the next cell of that row in the **Action** column.

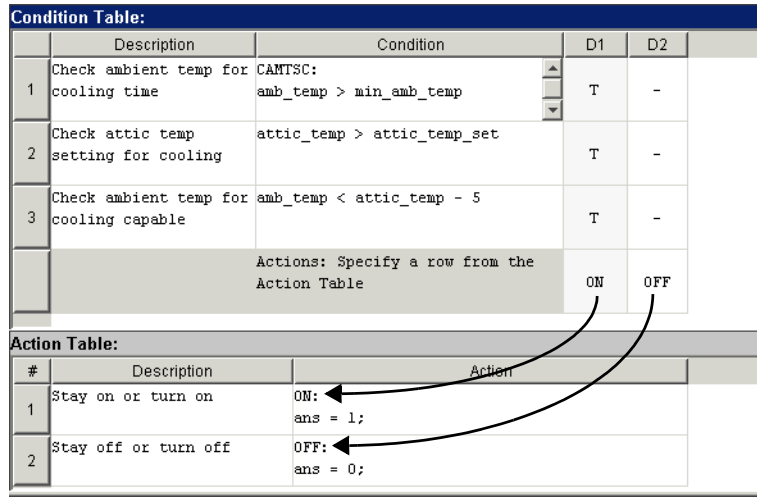
   You can begin the action with a label followed by a colon (`:`). Like condition labels, action labels must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (_).

   Specify truth table actions according to rules of Stateflow action language, as you would for a normal Stateflow action language statement. See "Actions" on page 2-18.

   You can use data defined for the Stateflow chart of the truth table to specify actions. You can also use data passed to the truth table function through the arguments and return value specified in the truth table function signature. See "Using Stateflow Data and Events in Truth Tables" on page 9-32 for more detail on using Stateflow data and events in truth table conditions and actions.

Note If you used row numbers to specify actions in step 1, make sure that you enter these actions in the matching rows of the **Action Table**.

The following example shows a completed truth table with actions specified for each possible decision outcome through the labels ON and OFF:



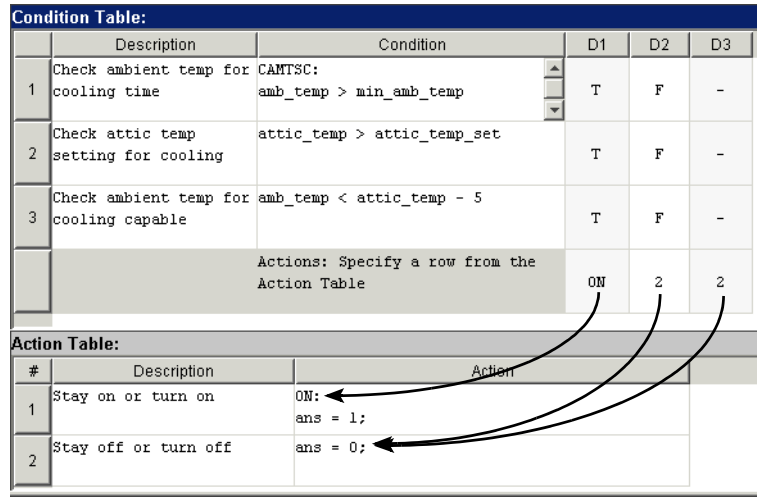
In this example, the first decision outcome is assigned the action `ans = 1`, which returns a true value for the function, indicating that the attic fan must remain on or be turned on. The second decision outcome is the default decision outcome for all remaining decision outcomes. Its action returns a Boolean value of false (0), which indicates that the fan remain off or be turned off.

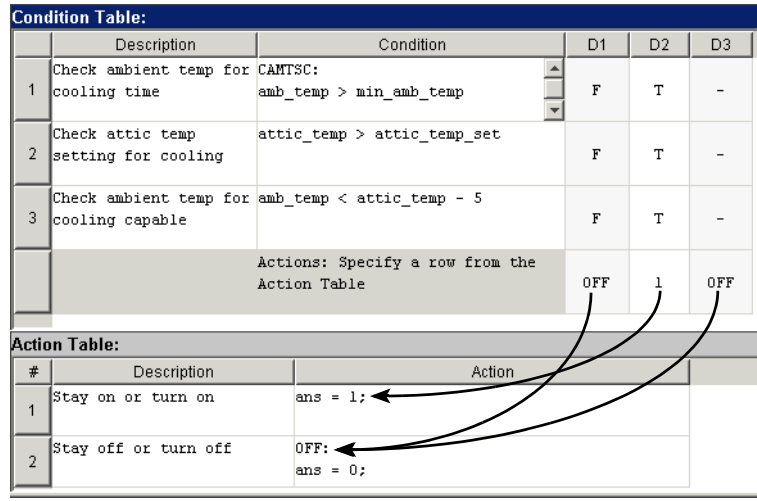
The following rules apply to entering and assigning actions to decision outcomes:

- You must specify an action for each decision outcome.

  Actions for decision outcomes are not optional. If you want to specify no action for a decision outcome, specify an empty action.
- You can specify multiple actions for a decision outcome with multiple specifiers separated by a comma.
- You can mix row number and label action specifiers interchangeably in any order.
- You can specify the same action for more than one decision outcome.

The following examples use the `check_temp` truth table example with an additional unneeded decision to demonstrate the use of action specifiers in the **Actions** row:



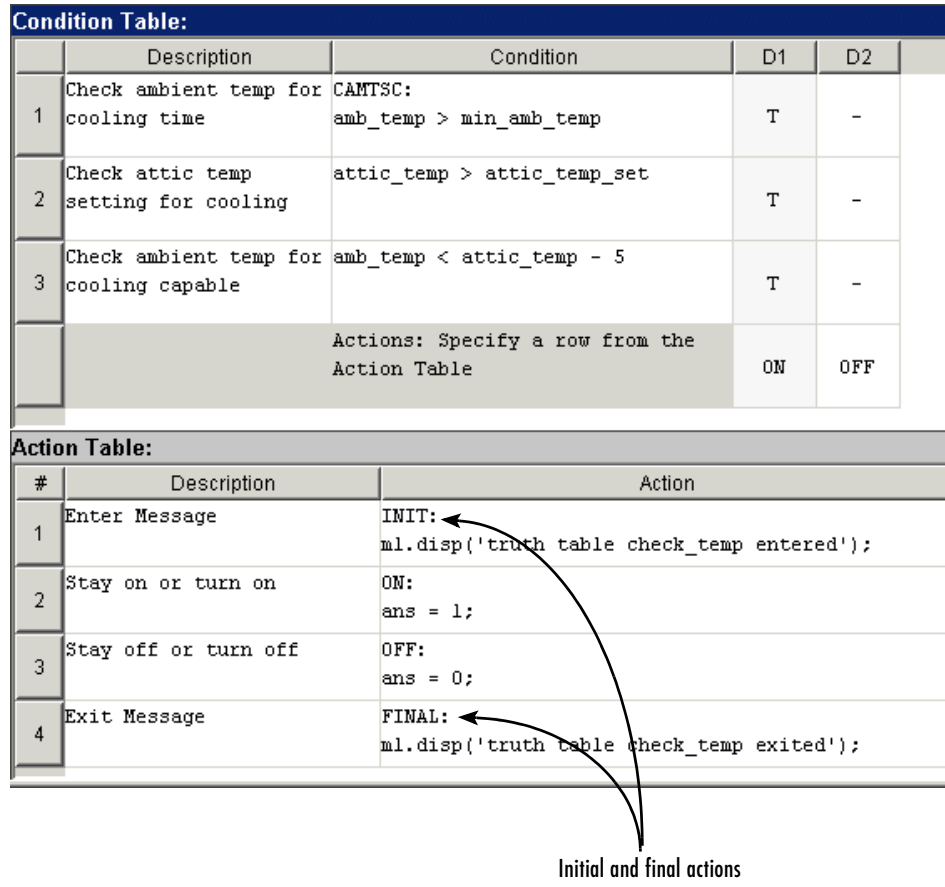


Special INIT and FINAL Actions

You can specify special actions that execute before and after the testing of decision outcomes. The action labeled INIT (uppercase, lowercase, or mixed) is executed before decision outcomes are tested. The action labeled FINAL

(uppercase, lowercase, or mixed) executes after the action for a realized decision outcome is executed but before the truth table function is exited.

The following example uses the INIT and FINAL labels to display diagnostic messages in the MATLAB Command Window.



Condition Table:

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-
		Actions: Specify a row from the Action Table	ON	OFF

Action Table:

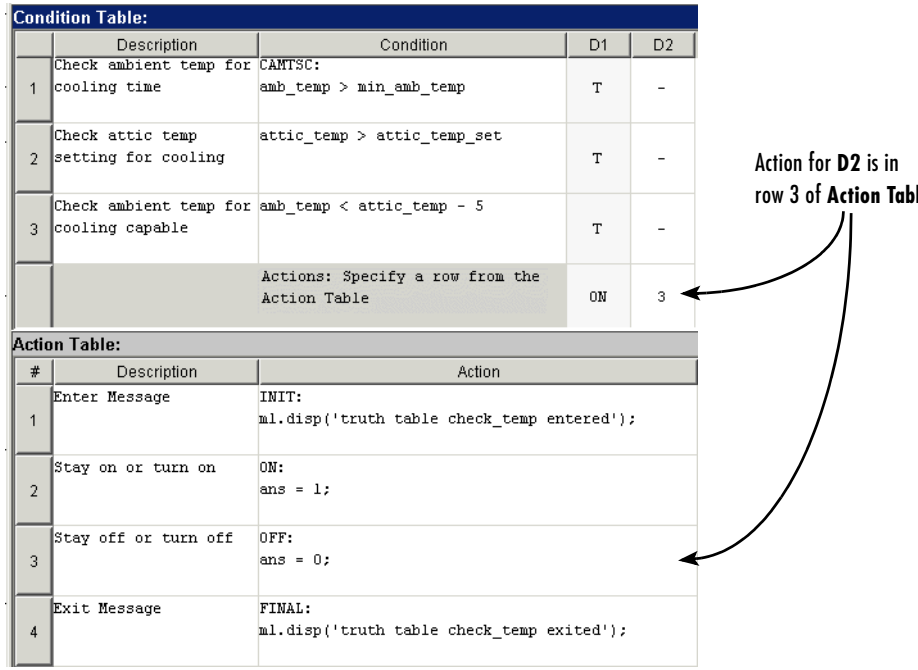
#	Description	Action
1	Enter Message	INIT: ml.disp('truth table check_temp entered');
2	Stay on or turn on	ON: ans = 1;
3	Stay off or turn off	OFF: ans = 0;
4	Exit Message	FINAL: ml.disp('truth table check_temp exited');

Note Even though the initial and final actions for the preceding truth table are shown in the first and last action rows, you can enter these actions in any order in the list of actions.

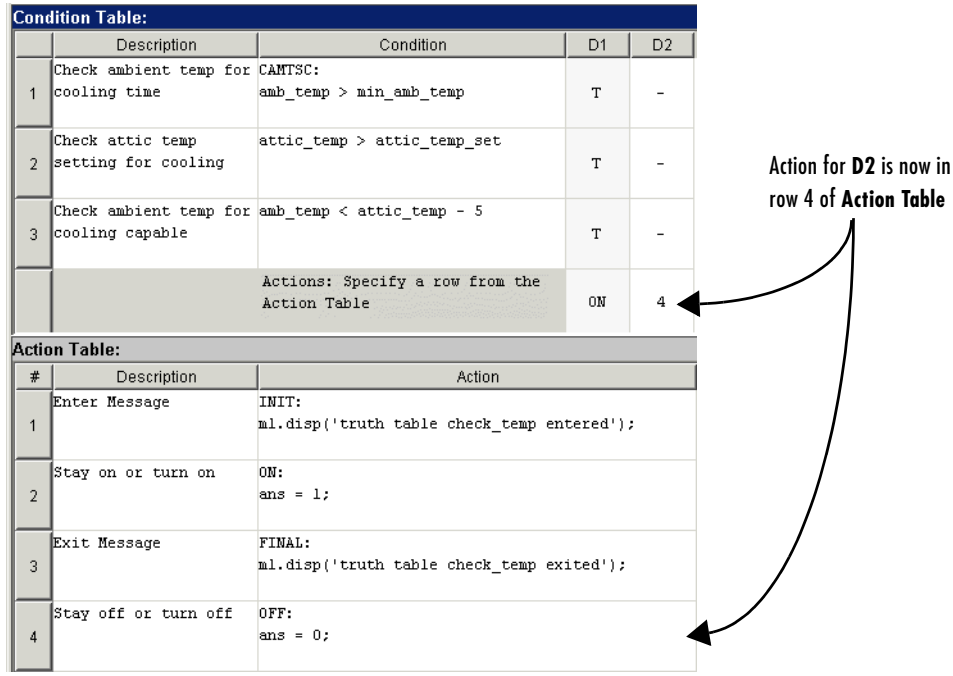
Numbered Row Tracking Feature

If you use **Action Table** row numbers to assign actions for decision outcomes, these rows are automatically tracked in the **Actions** row of the **Condition Table** and are readjusted if their row order is changed.

In the following example, decision outcome **D2** is assigned the action in row 3 of the **Action Table**.



If you now move row 4 ahead of row 3 in the **Action Table** by highlighting row 4 and selecting **Move Row Up** from the **Edit** menu, the **Actions** row action specifier for **D2** now indicates its action as row 4 of the **Action Table**.



Using Stateflow Data and Events in Truth Tables

When you compose conditions and actions (see "Entering Conditions" on page 9-23 and "Entering Actions for Decision Outcomes" on page 9-26), you have the following access to Stateflow data:

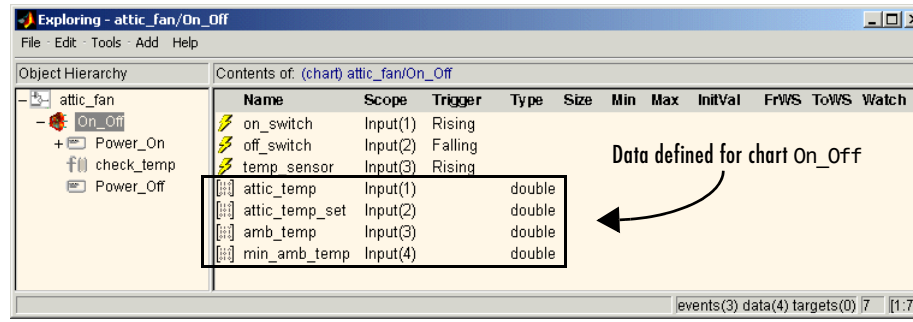
- You can use any parent or chart data to specify the conditions and actions of a truth table.

  A truth table can use any data defined for it, its parent, or higher parentage, including the chart. For example, if a truth table is defined at chart level, it can use chart local data, chart input and output data, and so on. If a truth table is defined in a state, it can also use data defined for the parent state and its parents, including the chart.

  You add data to a Stateflow chart through the **Add** menu in the Stateflow diagram window or through the Stateflow Explorer. Stateflow Explorer also

lets you define data for particular Stateflow objects such as states, boxes, and functions, including truth table functions.

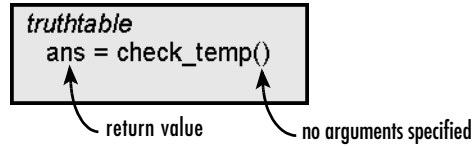
For example, the data `amb_temp`, `min_amb_temp`, `attic_temp`, and `attic_temp_set` are used in specifying the conditions for the truth table `check_temp`. These are all defined for the Stateflow chart `On_Off` (see "Creating a Model for the check_temp Truth Table" on page 9-60) that calls the truth table function `check_temp`, which is depicted in the following Stateflow Explorer window for that chart:



See "Adding Data to the Data Dictionary" on page 6-14 for details on adding data. See also "Your First Truth Table" on page 9-7 for an example of adding two data items to a Stateflow chart using the Explorer.

- You can use the arguments and return value for the truth table in the conditions and actions of the truth table.

  You can use the arguments and return value sent to the truth table function through its signature in conditions and actions for the truth table. For example, the data `ans` is the return value of the function signature for the truth table `check_temp` as it appears on the truth table box in the Stateflow diagram editor.



`ans` is used in the actions specified for the `check_temp` truth table. See the example in "Entering Actions for Decision Outcomes" on page 9-26.

- The actions in truth tables can broadcast events.

  The actions in a truth table can broadcast or send an event that is defined for the truth table, or for a parent, such as its chart. For example, the following action broadcasts an event `E` and then sets the data `ans` to 1:

```
2  Stay on or turn on        E;
                             ans = 1;
```

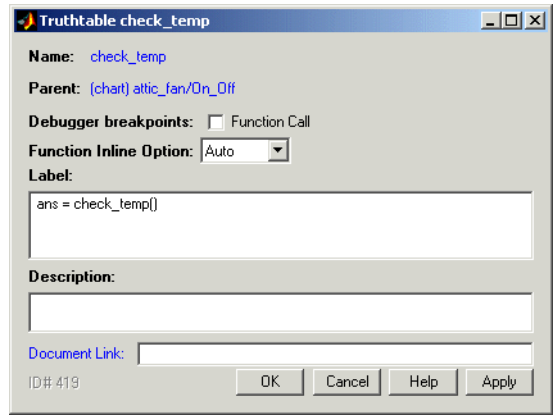
  You add events to a Stateflow chart through the **Add** menu in the Stateflow diagram window or through the Stateflow Explorer. However, the Stateflow Explorer also lets you define events scoped for the chart, or states, including truth table functions. See "Adding Events to the Data Dictionary" on page 6-2 for details on adding events.

Specifying Properties for a Truth Table

You set important properties for a truth table that are not available in the truth table editor (see "Editing a Truth Table" on page 9-20) in the **Truthtable Properties** dialog. Access this dialog as follows:

1. Right-click the truth table box in its Stateflow diagram.
2. Select **Properties** from the resulting pop-up menu.

   The **Truthtable Properties** dialog appears.



The fields in the **Truthtable Properties** dialog are as follows:

Field	Description
Name	The name of this truth table function. This is a read-only field; click this hypertext link to bring the truth table in the Stateflow diagram editor to the foreground.
Parent	Parent of the truth table function; a beginning / character indicates a Stateflow chart is the parent. This is a read-only field; click this hypertext link to bring the parent chart (in Simulink), or state, box, or graphical function (in Stateflow) to the foreground.
Debugger breakpoints	Select the **Function Call** check box to set a breakpoint where this function is called in action language. You can use this breakpoint during simulation to test your truth table. See “Debugging Truth Tables” on page 9-56 for more information.

Field	Description
Function Inline Option	This option controls the inlining of the truth table function diagram in generated code, through the following selections: • **Auto** Stateflow decides whether or not to inline the function based on an internal calculation. • **Inline** Stateflow inlines the function as long as it is not exported to other charts and is not part of a recursion. A recursion exists if the function calls itself either directly or indirectly through another called function. • **Function** The function is not inlined.
Label	The truth table's label specifying its signature. See "Creating a Graphical Function" on page 5-48 for more information.
Description	Text description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit/spec/data/speed.txt`.

Edit Operations in a Truth Table

The previous sections, "Introduction to Truth Tables" on page 9-2 and "Specifying Stateflow Truth Tables" on page 9-18, have mentioned several edit operations for the truth table editor such as the addition of rows and columns and the use of the Tab and arrow keys to navigate between table cells. The following topics completely describe all the edit operations available to you in the truth table editor:

- "Undoing and Redoing Truth Table Editor Operations" on page 9-37 — Shows you how to negate the effect of previous edit operations (undo) or redo the operations you negated (redo).
- "Adding Rows and Columns" on page 9-38 — Gives the procedures for appending and inserting rows and columns in the tables of the truth table editor.
- "Editing Individual Cells" on page 9-39 — Shows you how to edit and navigate individual text cells in the tables of the truth table editor.
- "Editing Rows and Columns" on page 9-41 — Gives the procedures for moving and deleting rows and columns, and changing row height in the tables of the truth table editor.
- "Optional Single Table Display" on page 9-44 — Shows you the convenience of displaying one or both tables in the truth table editor.
- "Searching and Replacing in Truth Tables" on page 9-45 — Gives you an example of using the Search & Replace tool to search for text and replace it in truth tables.
- "Making Printed and Online Copies of Truth Tables" on page 9-47 — Gives you the steps for producing printed or online file copies of your truth table.

Later, when you start constructing your own truth tables, you will need to troubleshoot them with the aid of truth table error checking. See "Error Checking for Truth Tables" on page 9-48 for details on the errors and warnings you receive.

Undoing and Redoing Truth Table Editor Operations

When you begin editing the tables in the truth table editor, it is important for you to know that you can undo and redo operations you perform in the truth table editor. When you undo an operation, you reverse the last edit operation

you performed. When you redo an operation, you restore the effects of an operation recently undone.

To undo and redo operations in the truth table editor, do the following:

1 Select the Undo toolbar button or press Ctrl+Z to undo the most recent operation.

2 Select the Redo toolbar button or press Ctrl+Y to redo the most recently undone operation.

To learn other edit operations you can perform in a truth table editor, see "Edit Operations in a Truth Table" on page 9-37.

Adding Rows and Columns

The first thing that you will probably want to do to an empty truth table is to add rows and columns to the **Condition Table** and rows to the **Action Table**. Later, as you fill in each row and column, you might need to insert new rows and columns. These and other row and column operations are described in the topics that follow.

To learn other edit operations you can perform in a truth table editor, see "Edit Operations in a Truth Table" on page 9-37.

Appending Rows to a Table

To add a new blank row to the end of a table in the truth table editor, do the following:

1 Make sure that the header of the table you want to add rows to is highlighted. If not, click a location in that table.

2 Select the Append Row toolbar button or select **Add Row** under the **Edit** menu.

A blank row is added to the bottom of the table.

Appending Decision Outcome Columns to the Condition Table

To add a decision column to the end of the **Condition Table** in the truth table editor, do the following:

1 Make sure that the header of the **Condition Table** is highlighted. If not, click a location in that table.

2 Select the Append Column toolbar button or select **Add Column** under the **Edit** menu.

A new blank column is added to the end of the table.

Inserting Rows in a Table

To insert a row above an existing row, do the following:

1 Right-click any cell in the existing row, including the row header.

2 From the resulting pop-up menu, select **Insert Row**.

A new blank row is added above the existing row.

Inserting Decision Outcome Columns in the Condition Table

To insert a new blank decision outcome column to the left of an existing decision outcome column, do the following:

1 Right-click any cell in the existing decision outcome column, including its column header.

2 From the resulting pop-up menu, select **Insert Column**.

A new blank column is added to the left of the existing column.

Editing Individual Cells

You edit individual cells to specify text descriptions, conditions, decision outcomes, and actions in a truth table. Operations for editing individual cells to enter text are described in the topics that follow.

To learn other edit operations you can perform in the truth table editor, see “Edit Operations in a Truth Table” on page 9-37.

Selecting Individual Cells for Editing

Click a cell in either the **Condition Table** or the **Action Table** to select it for editing.

- If you click a cell, its edge is highlighted and a blinking cursor is present for editing text in the cell.
- If you right-click a cell, its edge is highlighted and a pop-up menu appears for editing the contents of the cell and inserting a row above its row or a column to the left of its column.

To deselect a selected cell, click another table, cell, row, or column, or press the Esc key.

Keyboard Navigation Between Cells

- Use the Tab key to advance cell selection to the next cell on the right. This places the text cursor at the end of the text string in the cell.

  For the last cell in the row, the Tab key wraps around to first cell in the next row.
- Use Shift-Tab to select the next cell to the left.

  For the first cell in the row, Shift-Tab wraps around to last cell in the previous row.
- In the condition outcome cells that take only `T`, `F`, or `-` values, you can use the left, right, up, and down arrow keys to advance to another cell in any direction.

  In the cells of the Description, Condition, and Action columns, and the cells of the Action row, you use the left and right arrow keys to move the cursor to the left and right within the text.

Editing Text in Cells

- In the outcome cells that take only T, F, or - values, use the spacebar to step through the T, F, and - values.

  You can also use the backspace key to delete the current entry and then type a T, F, or - value.
- In the cells of the **Description** and **Condition** columns and the cells of the **Action** row and **Action Table**, edit text normally.

Deleting the Contents of a Cell

You can delete the contents of a single cell as follows:

1 Right-click the cell.

2 From the resulting pop-up menu, select **Delete Cell**.

Copying and Pasting the Contents of One Cell to Another

To copy or cut (copy and delete) the contents of a cell in the truth table editor, do the following:

1 Right-click the source cell.

2 Do one of the following:

- Select **Copy Cell** from the resulting pop-up menu to copy the cell contents.
- Select **Cut Cell** from the resulting pop-up menu to cut the cell contents.

You can also copy or cut the contents of a cell with the following:

1 Click the cell.

2 Do one of the following:

- Select the Copy toolbar button or, from the **Edit** menu, select **Copy**, to copy the contents.
- Select the Cut toolbar button or, from the **Edit** menu, select **Cut**, to cut the contents.

Paste the copied contents of a cell to another cell as follows:

1 Right-click a receiving cell.

2 From the resulting pop-up menu, select **Paste Cell**.

You can also paste the contents of the cell with the following:

1 Click the receiving cell.

2 Select the Paste toolbar button or, from the **Edit** menu, select **Paste**.

Editing Rows and Columns

This topic describes operations for editing complete rows and columns in the truth table editor.

To learn other edit operations you can perform in the truth table editor, see "Edit Operations in a Truth Table" on page 9-37.

Selecting Entire Rows and Columns for Editing

You select entire rows or columns to move, change, or delete them. To select an entire row for editing,

1 Click the header for a row or column.

   If you click the header for a row, the entire row is selected (highlighted).

   If you click the header for a decision outcome column in the **Condition Table**, the entire column is selected (highlighted), including its action.

To deselect a selected row or column, click another table, cell, row, or column.

Moving Rows and Columns

To move an entire row up or down in the order of rows in a truth table editor table, do the following:

1 Click the row header to select the row.

2 Do one of the following:

- Select the Move Row Up button or, from the **Edit** menu, select **Move Row Up**, to move the row up.
- Select the Move Row Down button or, from the **Edit** menu, select **Move Row Down**, to move the row down.

To move an entire column left or right in the order of columns, do the following:

1 Click the column header to select the column.

2 Do one of the following:

- Select the Move Column Left button , or from the **Edit** menu select **Move Column Right** to move the column left.
- Select the Move Column Right button or, from the Edit menu, select **Move Column Right**, to move the column right

Changing Row Height

The entries for conditions and actions can be quite long. When content exceeds row height of a cell, scroll bars appear on the right edge of the cell. However, you can increase the height of a row to see the complete contents of a cell as follows:

To change the height of all rows in a truth table editor table,

1. Click any location in the table, except a row header..
2. Do one of the following:
   - Select the Increase Row Height button or, from the **Edit** menu select **Increase Row Height**, to increase row height by one line space.
   - Select the Decrease Row Height button or, from the **Edit** menu select **Decrease Row Height**, to decrease row height by one line space.

To change the height of a particular row, click its row header and repeat the preceding step 2.

Copying and Pasting Rows and Columns

To copy and paste a row or column to another location in the table, do the following:

1. Right-click the row or column header.
2. From the resulting menu, select **Copy Row** or **Copy Column**.

   You can also select **Cut Row** or **Cut Column** to copy and delete the row or column.
3. Right-click the row below the new row location or right-click the column to the right of the new column location.
4. From the resulting pop-up menu, select **Paste Row** or **Paste Column**.

Deleting Rows and Columns

To delete an entire row or column in the truth table editor, do the following:

1. Right-click the row or column header.

2 From the resulting pop-up menu, select **Delete Row** or **Delete Column**.

You can also delete a row or column as follows:

1 Click the row or column header to highlight the entire row or column.

2 Press the Delete key.

You can compact a table by deleting its blank rows and columns as follows:

1 Click anywhere in the table to select it.

2 Select the Compact Table button , or from the **Tools** menu select **Compact**.

Row and Column Tooltip Identifiers

Stateflow gives you row and column header tooltips to aid you when you are editing a large truth table with many columns and/or rows. When you place your mouse cursor over the following row or column headers, a tooltip appears to identify it:

Table	Row or Column	ToolTip Display
Condition	Condition row	The condition entered for this row
Condition	Decision column (**D1**, **D2**,...)	The row or label entered for this decision in the Action row
Condition	**Actions** row	"Actions: specify a row from the Action Table"
Action	Any row	The description entered for this action

Optional Single Table Display

You can set the truth table editor to display only one or both of its two tables (**Condition Table** or **Action Table**) to give you a better view and more room to edit without scrolling.

- To display only one of two tables in a truth table, double-click the header of the table you want to display.
- To return to the display of both tables, double-click the header of the displayed table.

To learn other edit operations you can perform in the truth table editor, see "Edit Operations in a Truth Table" on page 9-37.

Searching and Replacing in Truth Tables

This topic teaches you how to use the Search & Replace tool to find and replace text in truth tables.

To learn other edit operations you can perform in the truth table editor, see "Edit Operations in a Truth Table" on page 9-37.

You can use the Search & Replace tool in Stateflow to search for text in the **Description**, **Condition**, and **Action** columns of a truth table and replace it with a substitute string. For example, you can search the model containing the truth table you built in "Your First Truth Table" on page 9-7 for the string `x is equal to` and replace it with the string `x equals` with the following procedure:

1. In the Stateflow diagram editor, select **Search & Replace** from the **Tools** menu.

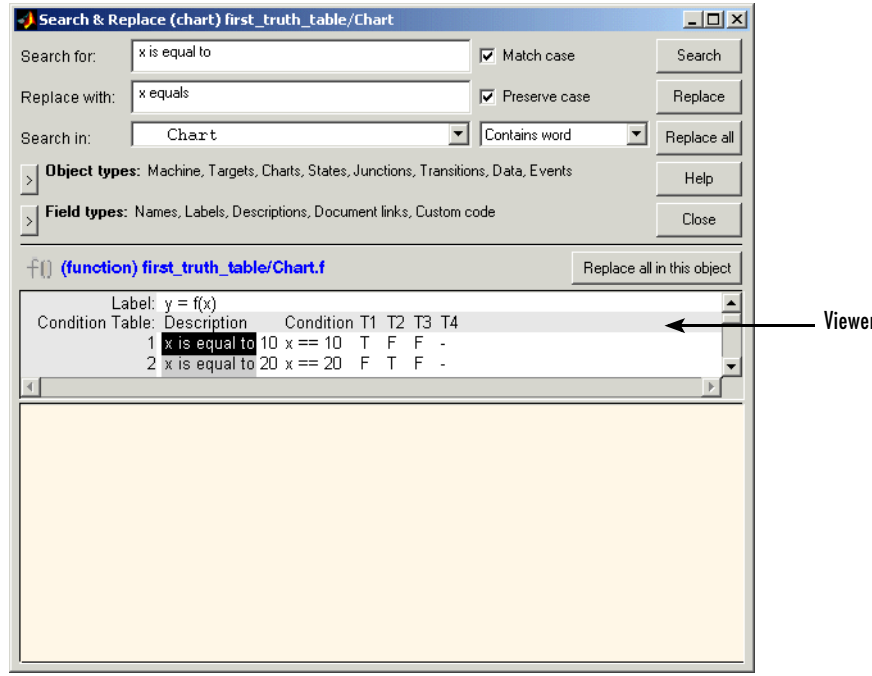
   The **Search & Replace** window appears.

2. In the **Search** field, enter the text `'x is equal to'`.

3. In the **Replace** field enter the text `'x equals'`.

4. Select the **Search** button.

   You should now see something like the following in the **Search & Replace** window:


Search & Replace (chart) first_truth_table/Chart
Search for:
x is equal to
Match case
Search
Replace with:
x equals
Preserve case
Replace
Search in:
Chart
Contains word
Replace all
Object types: Machine, Targets, Charts, States, Junctions, Transitions, Data, Events
Help
Field types: Names, Labels, Descriptions, Document links, Custom code
Close
(function) first_truth_table/Chart.f
Replace all in this object
Label: y = f(x)
Condition Table: Description Condition T1 T2 T3 T4
1 x is equal to 10 x == 10 T F F -
2 x is equal to 20 x == 20 F T F -
Viewer

Notice that in the Viewer pane of the **Search & Replace** window the first occurrence of the string `x is equal to` is highlighted normally and the remaining matches are highlighted lightly.

5 Select **Replace** to replace the first match with `'x equals'`.

6 Select **Replace All** to replace all matches with `'x equals'` in the model (not just the truth table).

Note The Search & Replace tool is fully documented in "The Stateflow Search & Replace Tool" on page 10-16.

Making Printed and Online Copies of Truth Tables

You can generate both printed and online copies of truth tables in the truth table editor. To make a printed copy of your truth table,

1. Select the Print button , or from the **File** menu select **Print**.
2. In the resulting **Print** dialog, select a printer.

To make an online viewable copy of your truth table,

1. From the **File** menu select **Export to HTML**.
2. In the resulting dialog window, browse for the location of the HTML file and enter its name in the **File name** field.
3. Select the **Save** button to save the HTML file.

   If you select **Save**, the HTML file automatically appears in the MATLAB online Help browser.

   If you select **Cancel** instead of **Save**, no HTML file is generated and the online Help browser does not appear.

To learn other edit operations you can perform in the truth table editor, see "Edit Operations in a Truth Table" on page 9-37.

Error Checking for Truth Tables

Once you have completely specified your truth tables (see "Specifying Stateflow Truth Tables" on page 9-18) you need to begin the process of debugging them. Stateflow runs its own diagnostics to check truth tables for syntax errors. See the following topics to find out when Stateflow checks truth tables for errors and a description and workaround for each error or warning:

- "When Stateflow Checks Truth Tables for Errors" on page 9-48 — Describes the user actions that cause error checking for a truth table.
- "Errors Detected During Error Checking" on page 9-49 — Lists and describes the errors that Stateflow detects for truth tables during error checking. Errors are conditions in your truth table that lead to undefined behavior.
- "Warnings Detected During Error Checking" on page 9-51 — Lists and describes the warnings that Stateflow detects for truth tables during parsing. Warnings point out conditions in your truth table that might lead to unexpected behavior but do not prevent execution of the truth table.

Continue debugging your truth tables through model simulation. See how Stateflow lets you monitor individual truth table execution steps in "Debugging Truth Tables" on page 9-56.

When Stateflow Checks Truth Tables for Errors

Stateflow checks truth tables for errors and warnings when you do any of the following:

- Select the Run Diagnostics button in the truth table editor toolbar
- Start simulation of the model with a new or modified truth table
- Start a Stateflow build of the model with a new or modified truth table
- Start a Stateflow rebuild of the model

The results of checking truth tables for errors are reported in the **Stateflow Builder** window, which also reports errors and warnings for parsing Stateflow diagrams. If there are no errors or warning, the **Stateflow Builder** window reports a message of success for all cases but simulation. See "Parsing Stateflow Diagrams" on page 11-27 for the mechanics of this window.

See a list and description of each error and warning that error checking tests for in "Errors Detected During Error Checking" on page 9-49 and "Warnings Detected During Error Checking" on page 9-51.

Errors Detected During Error Checking

The following errors are detected by Stateflow during the checking of truth tables for errors:

- `(Truth Table) Action Table, action` *`row_number`* `& action` *`row_number`*`: Duplicate action label '`*`action_label`*`'.`

  The specified rows in the **Action Table** contain actions labeled with the same label *`action_label`*. Each action must have a unique label.

- `(Truth Table) Condition Table, row` *`row_number`*`: Empty condition string.`

  No condition is specified for the condition in row *`row_number`* in the **Condition Table**.

- `(Truth Table) Condition Table, column (`*`column_number`*`): Default decision column (column containing all 'don't cares') must be the last column.`

  The default decision outcome column (see "Specifying Default Decision Outcomes" on page 9-26) is entered in column *column_number*, which is not the last decision outcome column. Since the default decision outcome column specifies all possible outcomes not specified by those columns to the left of it, no decision outcome columns beyond *column_number* can be executed in the generated diagram. Move the default decision outcome column to the last column position in the **Condition Table**.

- `(Truth Table) Condition Table, column` *`column_number`*`: No action is specified.`

  No action is specified for the decision outcome in column *`column_number`* in the **Condition Table**.

- `(Truth Table) Condition Table, column D<`*`column_number`*`>: Action '<`*`action_row_or_label`*`>' is not defined in Action Table.`

  The action designation *`<action_row_or_label>`* specifying an action for a decision outcome column in the **Condition Table** has no corresponding action in the **Action Table**.

- `(Truth Table) Condition Table, column D<column_number> row <row_number>: T/F/- cells cannot be empty.`

  The condition outcome in row `<row_number>` in the decision outcome in column `D<column_number>` in the **Condition Table** is empty. Only the entries `T`, `F`, or `-` are valid for a condition outcome.

- `(Truth Table) Underspecification found in Condition Table for following missing cases: cases`

  Displays the decision outcomes that are not specified in the **Condition Table** for which you might want to specify an action. A possible workaround is to specify a default decision outcome. See "Over- and Underspecified Truth Tables" on page 9-52.

  By default, an underspecified truth table produces an error. You can choose to report this condition as an error or warning, or not at all, by selecting **Diagnostics** from the truth table menu followed by **Underspecified** in the resulting submenu followed by **Error**, **Warning**, or **None** from the resulting submenu.

- `(Truth Table) Overspecification found in Condition Table: Column D<column_number> is overspecified by columns i, j,....`

  The decision outcomes specified in column `D<column_number>` in the **Condition Table** are already specified in the decision outcomes specified in previous columns D*i*, D*j*,.... This means that the action specified for column D`<column_number>` will never be executed. See "Over- and Underspecified Truth Tables" on page 9-52.

  By default, an overspecified truth table produces an error. You can choose to report this condition as an error or warning, or not at all, by selecting **Diagnostics** from the truth table menu followed by **Overspecified** in the resulting submenu followed by **Error**, **Warning**, or **None** from the resulting submenu.

- `'(Truth Table) Condition Table, user label 'user_label' in condition n collides with an autocreated label for condition m.`

  Truth tables use condition labels to name temporary data belonging to the truth table to store the result of a condition. For example, if the third condition (row 3 of the **Condition Table**) has no label specified for it, temporary data `c3` is autogenerated for it internally. If you specify the same label (that is, 'c3') explicitly for an earlier condition (say row 2), a data

conflict occurs. The workaround is to use a condition label with a different form than c*n*, where *n* is the row of the condition column.

- `(Truth Table) Condition Table, user label '`*`user_label`*`' in condition` *`n`* `collides with a user created data #`*`id`*.

  You specified data for the truth table function with the same name as the label *user_label* that you gave to condition *n*, where *n* is the row number of the condition in the **Condition Table**. The number *id* is an internal reference to this data. The workaround is to name the data or the label differently.

- `'(Truth Table) Condition Table, autocreated label 'c`*`n`*`' for condition` *`n`* `collides with a user-created data #`*`id`*.

  Truth tables use condition labels to name temporary data belonging to the truth table to store the result of a condition. For example, if the third condition (row 3 of the **Condition Table**) has no label specified for it, temporary data `c3` is autogenerated for it internally. If you specify Stateflow data `c3` for the truth table function, a data conflict occurs. The workaround is to change the name of the conflicting data from the c*n* format.

Warnings Detected During Error Checking

The following warnings are detected by Stateflow during the checking of truth tables for errors:

- `(Truth Table) Action Table, action (<`*`row_number`*`>): Unreferred actions by Condition Table.`

  An action is specified in row <*row_number*> of the **Action Table** that is not referenced by a decision outcome column in the **Condition Table**.

- `(Truth Table) Condition Table contains no conditions or decisions.`

  The **Condition Table** is empty.

- `(Truth Table) Condition Table, column (<`*`column_number`*`>): Action 'INIT' is reserved for user input truth table initialization. It's being called inside truth table body.`

  The action label `'INIT'` is entered in column number *column_number* in the **Condition Table**. This label is reserved for a special initial action.

- `(Truth Table) Condition Table, column (<column_number>): Action 'FINAL' is reserved for user input truth table finalization. It's being called inside truth table body.`

  The action label `'FINAL'` is entered in column number *column_number* in the **Condition Table**. This label is reserved for a special final action.
- `(Truth Table) Underspecification found in Condition Table for following missing cases: cases`

  By default this is reported as an error unless specified as a warning. See the same message in "Errors Detected During Error Checking" on page 9-49 for a description.
- `(Truth Table) Overspecification found in Condition Table: Column <column_number> is overspecified by columns i, j,....`

  By default this is reported as an error unless specified as a warning. See the same message in "Errors Detected During Error Checking" on page 9-49 for a description.

Over- and Underspecified Truth Tables

Over- and underspecified truth tables are two special error conditions that truth table error checking tests for. See the topics that follow for a more complete understanding of these errors.

For a complete list of errors and warnings, including a description and possible workaround, see "Errors Detected During Error Checking" on page 9-49 and "Warnings Detected During Error Checking" on page 9-51./

Overspecified Truth Table

An overspecified truth table contains at least one decision outcome that will never be executed because it is already specified in a previous decision outcome in the **Condition Table**.

The following is the **Condition Table** of an overspecified truth table:

Condition Table:

	Description	Condition	D1	D2	D3
1	Condition C1	C1: x == 0	F	T	-
2	Condition C2	C2: y == 0	T	-	T
3	Condition C3	C3: z == 0	T	T	T
		Actions: Specify a row from the Action Table	A1	A2	A3

The decision outcome in column **D3** (-TT) actually specifies the decision outcomes FTT and TTT that have already been specified by decisions **D1** (FTT) and **D2** (TTT and TFT). Therefore column **D3** is an overspecification.

The following is the **Condition Table** of a truth table that appears overspecified but is not:

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	F	T	T	-
2	Condition C2	C2: y == 0	T	F	T	T
3	Condition C3	C3: z == 0	T	T	F	T
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

In this case, the decision outcome D4 specifies two decision outcomes (TTT and FTT). The second of these is specified by decision outcome **D1**. However, the first is not specified in a previous column and therefore, this **Condition Table** is not overspecified.

Underspecified Truth Table

An underspecified truth table lacks one or more possible decision outcomes that might require an action to avoid undefined behavior.

The following is the **Condition Table** of an underspecified truth table:

Condition Table:

	Description	Condition	D1	D2	D3
1	Condition C1	C1: x == 0	T	T	F
2	Condition C2	C2: y == 0	T	F	T
3	Condition C3	C3: z == 0	F	T	T
		Actions: Specify a row from the Action Table	A1	A2	A3

Complete coverage of this truth table requires a **Condition Table** with every possible decision outcome, like the following example:

Condition Table:

	Description	Condition	D1	D2	D3	D4	D5	D6	D7	D8
1	Condition C1	C1: x == 0	T	T	T	F	F	T	F	F
2	Condition C2	C2: y == 0	T	T	F	T	F	F	T	F
3	Condition C3	C3: z == 0	T	F	T	T	F	F	F	T
		Actions: Specify a row from the Action Table	A1	A2	A3	A4	A5	A6	A7	A8

A possible workaround is to specify an action for all other possible decision outcomes through a default decision outcome, as in the following example:

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	T	T	-
2	Condition C2	C2: y == 0	T	T	F	-
3	Condition C3	C3: z == 0	T	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	DA

See “Specifying Default Decision Outcomes” on page 9-26.

Debugging Truth Tables

When you choose to simulate your model, Stateflow checks the truth tables automatically for syntactical errors if they have changed since the last simulation. See “Error Checking for Truth Tables” on page 9-48 for descriptions and workarounds for the errors and warnings you receive.

Later, during simulation of your model, you can debug each execution step of a truth table using the Stateflow Debugger. See the following topics to learn the procedure for debugging a truth table during simulation and an example of what you might see during debugging of the `check_temp` truth table:

- “How to Debug a Truth Table During Simulation” on page 9-56 — Shows you the procedure to monitor the execution of individual conditions, decisions, and actions during simulation using the Stateflow Debugger.
- “Debugging the check_temp Truth Table” on page 9-57 — Gives you a step-by-step example of what you see when you monitor the `check_temp` truth table during simulation with the Stateflow Debugger.
- “Creating a Model for the check_temp Truth Table” on page 9-60 — Provides you with a Simulink model you can use to simulate the `check_temp` truth table.

During simulation of your truth tables, you hope to exercise every part of them to check them for run-time errors. To be sure that every part of a truth table has been executed during simulation testing, you’ll need to know what parts of the truth table have executed and what parts have not. See “Model Coverage for Truth Tables” on page 9-64 to continue.

How to Debug a Truth Table During Simulation

To simulate the individual operation steps of a truth table, do the following:

1 Set a breakpoint for the call made to the truth table.

  Select the **Debugger breakpoints** property in the **Truthtable Properties** dialog. See “Specifying Properties for a Truth Table” on page 9-34 for a description of this property.

2 Select the Debug button in the Stateflow diagram editor toolbar to start the Stateflow Debugger window.

   You can also select **Debug** from the **Tools** menu in the Stateflow diagram editor.

3 From the Stateflow Debugger select the **Start** button to begin simulation of your model.

4 Wait until the breakpoint for the call to the truth table is reached.

   When the breakpoint is encountered, the **Start** button in the Stateflow Debugger turns to the **Continue** button.

5 Select the **Step** button in the debugger window to advance simulation one step through the truth table.

6 Continue using the **Step** button to step through truth table execution.

   Observe the executing rows and columns of the truth table that appear highlighted during execution.

See the debugging process for the example `check_temp` truth table in "Debugging the check_temp Truth Table" on page 9-57.

Debugging the check_temp Truth Table

The example that follows shows the display of executing truth table rows and columns that appear during simulation of the `check_temp` truth table using the Stateflow debugger. For instructions on assembling the complete model to test this yourself, see "Creating a Model for the check_temp Truth Table" on page 9-60.

The following example simulation of the `check_temp` truth table begins when you step through the simulating truth table as specified in the previous topic, "How to Debug a Truth Table During Simulation" on page 9-56.

1 First, the `INIT` action is executed.

Action Table:

	Description	Action
1	Enter Message	INIT: ml.disp('truth table check_temp entered');
2	Stay on or turn on	ON: ans = 1;
3	Stay off or turn off	OFF: ans = 0;
4	Exit Message	FINAL: ml.disp('truth table check_temp exited');

2 Next, each condition is evaluated.

Condition Table:

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-

Condition Table:

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-

Condition Table:

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-

3 Next, each decision outcome is tested until one is found to be true. In this case, the first decision outcome is true.

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-
		Actions: Specify a row from the Action Table	ON	OFF

4 Because the first decision outcome is true, the action for its decision outcome is executed.

Action Table:

	Description	Action
1	Enter Message	INIT: ml.disp('truth table check_temp entered');
2	Stay on or turn on	ON: ans = 1;
3	Stay off or turn off	OFF: ans = 0;
4	Exit Message	FINAL: ml.disp('truth table check_temp exited');

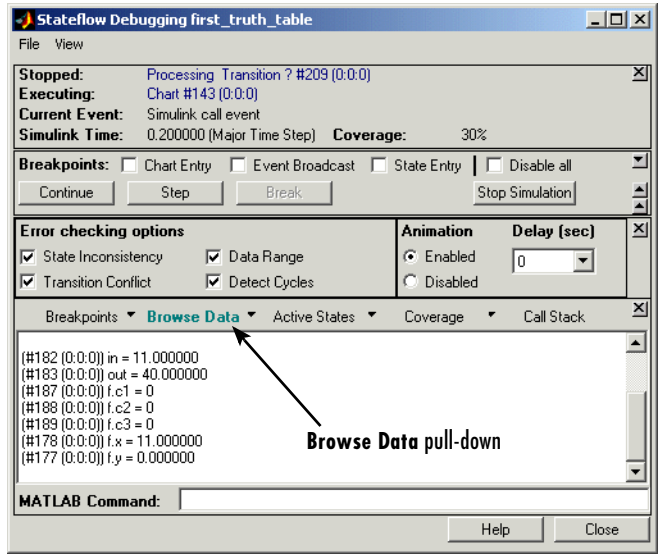
5 Finally, the `FINAL` action is executed.

Action Table:

	Description	Action
1	Enter Message	INIT: ml.disp('truth table check_temp entered');
2	Stay on or turn on	ON: ans = 1;
3	Stay off or turn off	OFF: ans = 0;
4	Exit Message	FINAL: ml.disp('truth table check_temp exited');

On the Debugger window you can also monitor Stateflow data by selecting an option under the **Browse Data** pull-down. For example, while simulating the "your first truth table" model, selecting the **All Data (Current Chart)** option

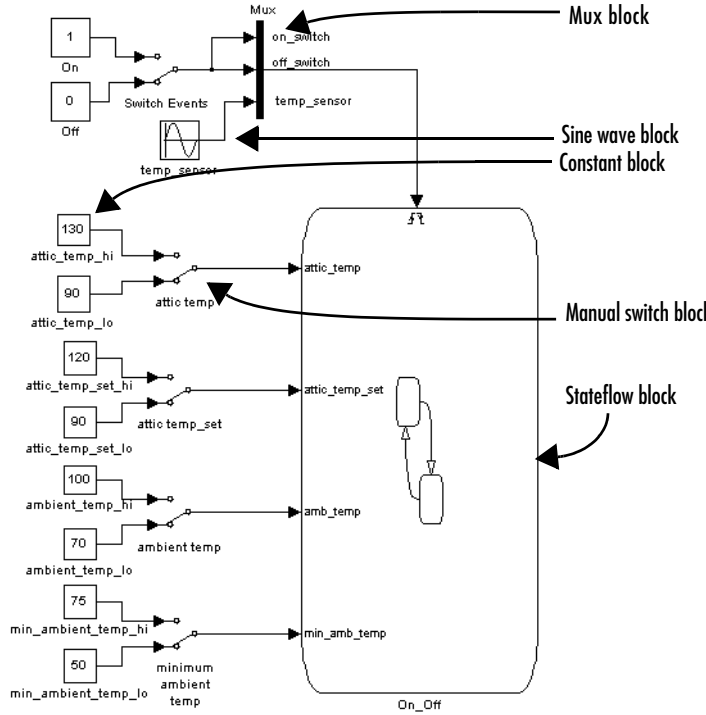
from the **Browse Data** pull-down produces the following continuously updated display:


Stateflow Debugging first_truth_table
File View
Stopped: Processing Transition ? #209 (0:0:0)
Executing: Chart #143 (0:0:0)
Current Event: Simulink call event
Simulink Time: 0.200000 (Major Time Step) Coverage: 30%
Breakpoints: Chart Entry Event Broadcast State Entry Disable all
Continue Step Break Stop Simulation
Error checking options
State Inconsistency Data Range
Transition Conflict Detect Cycles
Animation Delay (sec)
Enabled 0
Disabled
Breakpoints Browse Data Active States Coverage Call Stack
(#182 (0:0:0)) in = 11.000000
(#183 (0:0:0)) out = 40.000000
(#187 (0:0:0)) f.c1 = 0
(#188 (0:0:0)) f.c2 = 0
(#189 (0:0:0)) f.c3 = 0
(#178 (0:0:0)) f.x = 11.000000
(#177 (0:0:0)) f.y = 0.000000
Browse Data pull-down
MATLAB Command:
Help Close

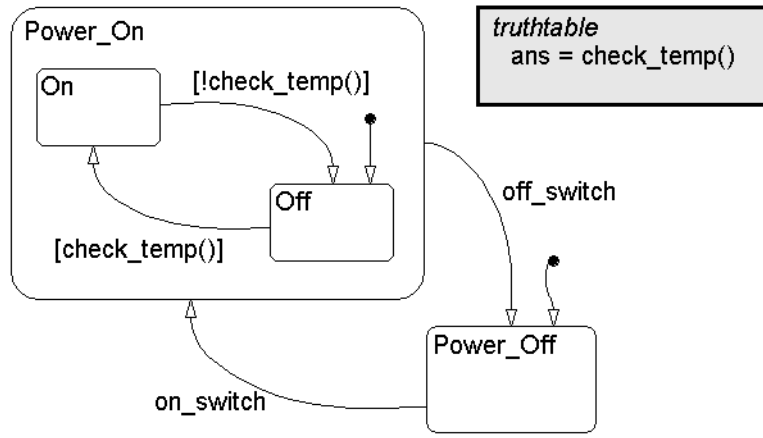
Creating a Model for the check_temp Truth Table

If you would like to simulate the `check_temp` truth table example as shown in "Debugging the check_temp Truth Table" on page 9-57, use the following procedure to build and test a model containing the `check_temp` truth table:

1 Build the following model in Simulink:

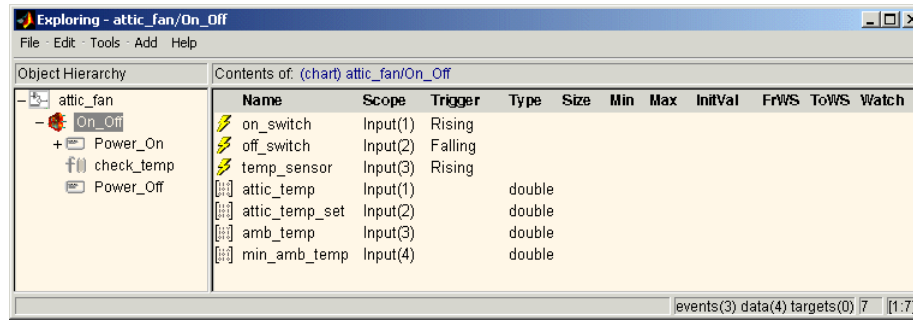


2 Specify the Stateflow block On_Off with the following Stateflow diagram:



This is the same diagram used in “Calling Truth Table Functions in Stateflow” on page 9-16.

3 Use Stateflow Explorer to specify the following events and data for the On_Off chart.



Make sure that the **Input from Simulink** events are specified in the order given. If you need to correct this, you can click and drag an event row to a location above or below another event row in the Explorer.

4 Use the truth table displayed in "Entering Actions for Decision Outcomes" on page 9-26.

   You might want to add special INIT and FINAL actions of your own or use the ones in "Special INIT and FINAL Actions" on page 9-29.

5 During simulation, do the following:

   - To test the `check_temp` truth table, be sure to follow the procedure given in "How to Debug a Truth Table During Simulation" on page 9-56.
   - Flip (double-click) the `Switch Events` manual switch between inputs `On` and `Off` to send `on_switch` and `off_switch` events to the `On_Off` chart.

     Notice that `on_switch` is a rising trigger and `off_switch` is a falling trigger. When the input to `Switch Events` rises from 0 to 1, this sends an `on_switch` event to the `On_Off` chart. Similarly, when the input falls from 1 to 0, this sends an `off_switch` event to the chart.

     The events `on_switch` and `off_switch` change the active state of the `On_Off` chart between the `Power_Off` and `Power_On` states. Sending these events turns the power to the fan on or off.
   - When the `On` state becomes active, change the values for the data inputs `attic_temp`, `attic_temp_set`, `ambient_temp`, and `minimum_ambient_temp`, using the manual switches that control their values.

     The sine wave input to the `temp_sensor` event causes an event to occur every second. When this event occurs while the `power_On` state is active, Stateflow seeks a valid transition between the `On` and `Off` states. This causes the check_temp truth table to be reevaluated to test the condition on the connecting transition.

Model Coverage for Truth Tables

Stateflow reports model coverages for truth tables as a function. This section describes the model coverage for truth table functions. For a more complete understanding of model coverage in Stateflow, see “Stateflow Chart Model Coverage” on page 12-26.

See the following topics for a description of model coverage for an example truth table:

- “Example Model Coverage Report” on page 9-64 — Gives an example model coverage report for a truth table.
- “Description of Coverages for the Example Report” on page 9-66 — Describes the individual coverages in the example Model Coverage report for a truth table.

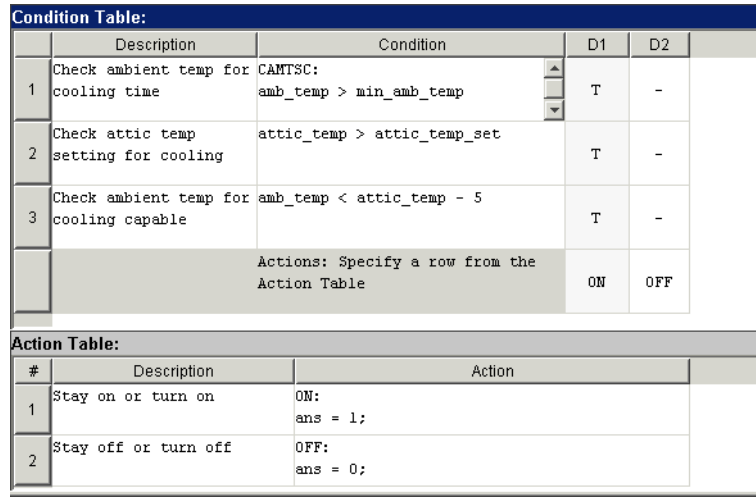
Note The Model Coverage tool is available to you if have the Simulink Performance Tools license.

To fully understand the model coverage for your truth tables, it is important for you to know how truth table functions are realized by Stateflow. See “How Stateflow Realizes Truth Tables” on page 9-68.

Example Model Coverage Report

You generate model coverage reports during simulation by specifying them in Simulink and then simulating your model. When simulation ends, a model coverage report appears in a browser window. See “Making Model Coverage Reports” on page 12-26 for information on how to set this up.

In this example, the following `check_temp` truth table, introduced in “Editing a Truth Table” on page 9-20, is tested during simulation.



Condition Table:

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-
		Actions: Specify a row from the Action Table	ON	OFF

Action Table:

#	Description	Action
1	Stay on or turn on	ON: ans = 1;
2	Stay off or turn off	OFF: ans = 0;

The following is the part of the model coverage report covering `check_temp`:

4. Function "check_temp"

Parent: attic_fan/On_Off

Metric	Coverage (this object)	Coverage (inc. descendents)
Cyclomatic Complexity	0	3
Decision (D1)	NA	100% (2/2) decision outcomes
Condition (C1)	NA	83% (5/6) condition outcomes
MCDC (C1)	NA	67% (2/3) conditions reversed the outcome

Predicate table analysis (missing values are in parentheses)

Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T (ok)	-
Check attic temp setting for cooling	attic_temp > attic_temp_set	T (ok)	-
Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T (F)	-
	Actions	ON (ok)	OFF

Description of Coverages for the Example Report

Coverage for the truth table function itself (in the **Coverage (this object)** column) shows no valid coverage values because there is no decision involved in the container object for the truth table function.

The logic of the truth table is implemented internally in the transitions of the graphical function generated for the truth table (see "How Stateflow Realizes Truth Tables" on page 9-68). These transitions are descendents of the function and contain the decisions and conditions of the truth table. Coverage for the descendents (in the **Coverage (inc. descendents)** column) includes coverage for the specified conditions and decisions that are tested when the truth table function is called. In the case of the `check_temp` truth table, the only decision outcome covered is the single **D1** decision outcome.

Note Because all logic leading to taking a default decision outcome is based on a false outcome for all preceding decision outcomes, no logic is required for the default decision outcome and it receives no model coverage.

Coverages for the descendent conditions and the **D1** decision of the `check_temp` truth table function are as follows:

- Decision coverage for the **D1** decision is 100% since this decision was tested both true and false during simulation.
- Condition coverage for the three conditions of the **D1** decision indicate that 5 of 6 T/F values were tested.

  This result is indicated in the table, which shows that the last condition received partial condition coverage by not evaluating to false (F) during simulation. The missing occurrence of the false (F) condition value is indicated by the appearance of a red `F` character for that condition in the **D1** decision outcome column. Since each condition can have an outcome value of `T` or `F`, 3 conditions can have 6 possible values. During simulation, only 5 of 6 were tested.
- MCDC coverage is 2 of 3 (67%) owing to the two reversals of the **D1** decision outcome by changing condition 1 from true to false and by changing condition 2 from true to false.

  MCDC coverage looks for decision reversals that occur because one condition value changes from `T` to `F` or from `F` to `T`. The **D1** decision outcome reverses

when any of the conditions changes from `T` to `F`. This means that the outcomes `FTT`, `TFT`, and `TTF` reverse this decision outcome by a change in the value of one condition.

The top two conditions tested both true (`T`) and false (`F`) with a resulting reversal in the decision outcome from true (`T`) to false (`F`). However, the bottom condition tested only to a true (`T`) outcome but no false (`F`) outcome (appearance of red `F` character). Therefore, only two of a possible three reversals were observed and coverage is 2/3 = 67%.

- The `(ok)` next to the `ON` action label indicates that its decision outcome realized both true (`T`) and false (`F`) during simulation. Since the default decision outcome is based on no logic of its own, it does not receive the `(ok)` mark.

How Stateflow Realizes Truth Tables

Stateflow realizes the logical behavior specified in a truth table by generating a graphical function. See the following sections if it is important for you to know the actual mechanics of truth table generation in Stateflow:

- "When Stateflow Generates Truth Table Functions" on page 9-68 — Tells you when Stateflow generates a graphical function for a truth table.
- "How to See the Generated Graphical Function" on page 9-68 — Tells you how to view the generated graphical function for a truth table.
- "How Stateflow Generates Truth Tables" on page 9-69 — Takes a particular example truth table and describes the structure of the resulting graphical function.

When Stateflow Generates Truth Table Functions

Stateflow generates the graphical function for a newly specified truth table when you simulate your model. If your truth table changes, its graphical function is regenerated when you choose to simulate your model. If your truth table remains unchanged, choosing to simulate does not regenerate a new graphical function.

You can see the generated graphical function for a truth table in the Stateflow diagram editor. See "How to See the Generated Graphical Function" on page 9-68.

How to See the Generated Graphical Function

Once you're sure that Stateflow has generated a graphical function for your truth table (see "When Stateflow Generates Truth Table Functions" on page 9-68) you can see the generated function for a truth table as follows:

1. Right-click the truth table box in its Stateflow diagram.
2. Select **View Contents** from the resulting pop-up menu.

The truth table graphical function appears with a shaded background to indicate that it is not editable. See "How Stateflow Generates Truth Tables" on page 9-69 for an example.

How Stateflow Generates Truth Tables

This topic shows you through an example how the logic of a truth table translates into an autogenerated graphical function. See “How to See the Generated Graphical Function” on page 9-68

The following example truth table has three conditions, four decision outcomes and actions, and initial and final actions:

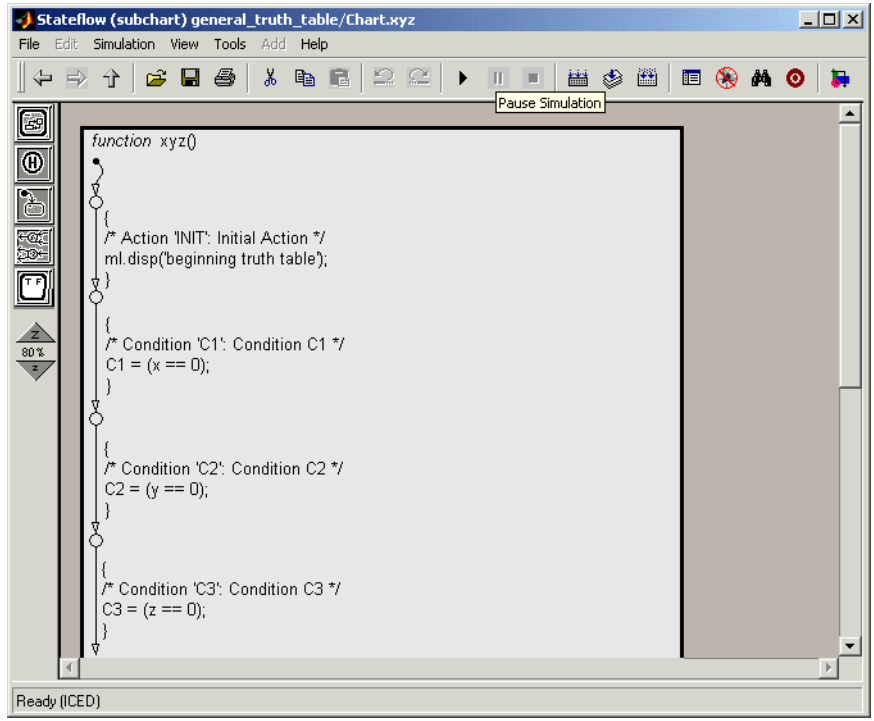
Condition Table:

	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	F	F	
2	Condition C2	C2: y == 0	-	T	F	
3	Condition C3	C3: z == 0	-	-	T	
	Enter Row number for associated action	Action number:	A1	A2	A3	DA

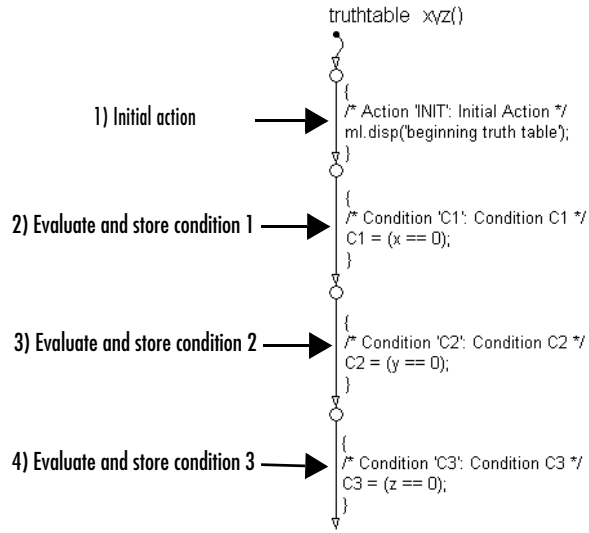
Action Table:

#	Description	Action
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Default Action	DA: x = 0; y = 0; z = 0;
6	Final Action	FINAL: ml.disp('ending truth table');

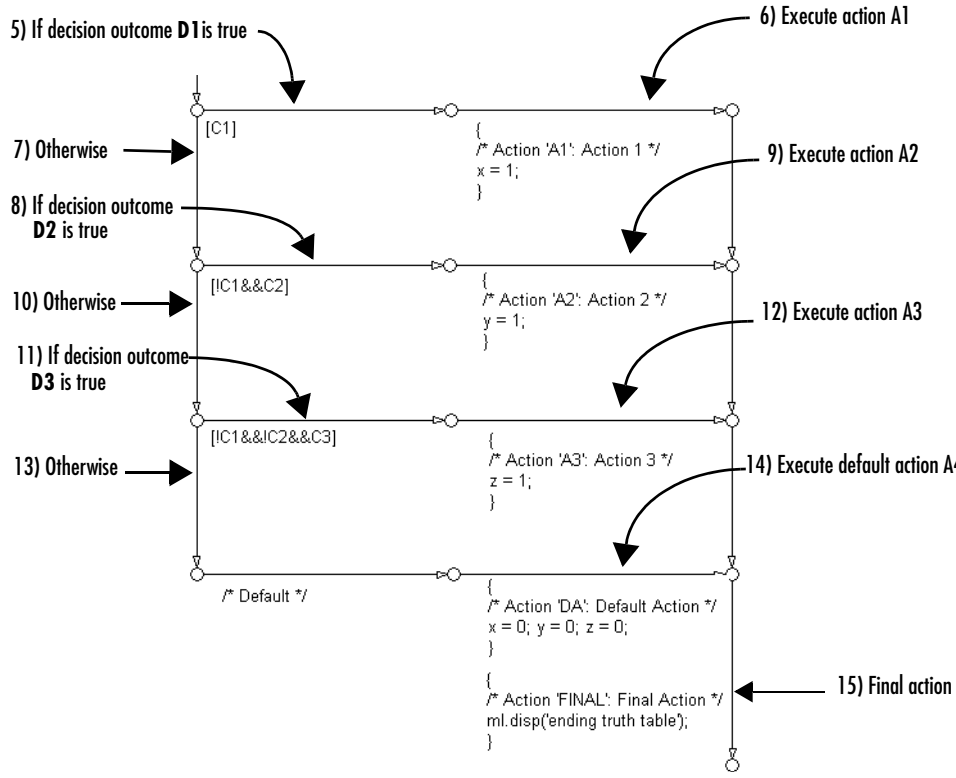
Stateflow generates the following graphical function for the preceding truth table:



In the generated function, Stateflow uses the top half of the generated function to perform the initial actions and compute and store the results of each evaluated condition in temporary variables that are automatically created when the model is generated.



The stored values for the conditions are then used in the bottom half of the function to make decisions on which action to perform.



Each decision is implemented as a fork from a connective junction with one of two possible paths:

- A transition segment with a decision followed by a segment with the consequent action specified as a condition action that leads to the termination of the flow diagram
- A transition segment with no condition or action that flows to the next fork for an evaluation of the next decision

This arrangement continues for the first decision (column1) through the remaining decisions in left to right column order, until the last decision. When a specified decision outcome is matched, the action specified for that decision is executed as a condition action of its transition segment and the flow graph terminates after the final action is executed. This means that one and only one

action results for a call to a truth table graphical function. This also implies that no data dependencies are possible between different decisions.

10

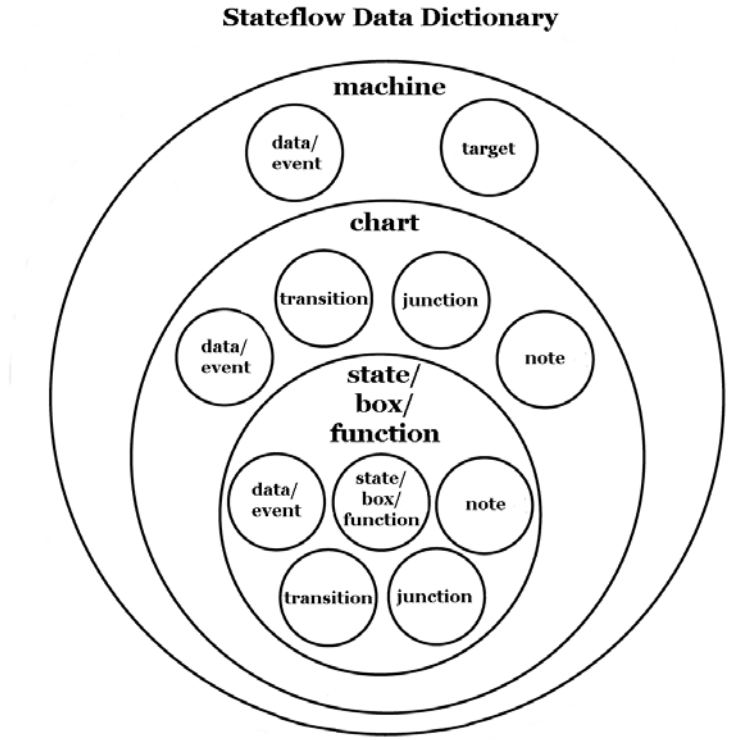
Exploring and Searching Charts

Stateflow provides you with tools for searching for objects and replacing them with others. Learn how to search and replace objects in Stateflow in the following sections:

Overview of the Stateflow Machine (p. 10-2)	Describes the Stateflow machine, the highest level in the Stateflow hierarchy, and the hierarchy of graphical and nongraphical objects within it.
The Stateflow Explorer Tool (p. 10-3)	Describes the Stateflow Explorer, a powerful tool for displaying, modifying, and creating data and event objects for any parent object in Stateflow. The Explorer also displays, modifies, and creates targets for the Stateflow machine.
Stateflow Explorer Operations (p. 10-8)	Describes the large number of operations you can perform on any Stateflow object in the Stateflow Explorer.
The Stateflow Search & Replace Tool (p. 10-16)	Stateflow Search & Replace tool searches for and replaces text belonging to objects in Stateflow charts.
The Stateflow Finder Tool (p. 10-31)	Stateflow provides the Stateflow Finder tool on platforms that do not support the Simulink Find tool. This section describes how you use the Stateflow Finder tool to search for objects in Stateflow.

Overview of the Stateflow Machine

The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of graphical and nongraphical objects. The data dictionary is the repository for all Stateflow objects.



You can use the tools described in the following sections to browse and make changes to data dictionary objects.

- "The Stateflow Explorer Tool" on page 10-3
- "The Stateflow Search & Replace Tool" on page 10-16
- "The Stateflow Finder Tool" on page 10-31

The Stateflow Explorer Tool

The Stateflow Explorer displays any defined event or data objects within an object hierarchy where machines, charts, states, boxes, and graphic functions are potential parents. You can use the Stateflow Explorer to create, modify, and delete event and data objects for any parent object. You can also display, create, modify, and delete target objects in the Stateflow Explorer (only a Stateflow machine can parent target objects).

See the following topics describing the Stateflow Explorer:

- "Opening Stateflow Explorer" on page 10-3 — Tells you how to open the Stateflow Explorer in the Stateflow diagram editor.
- "Explorer Main Window" on page 10-4 — Describes the parts of Explorer window.
- "Objects in the Explorer" on page 10-5 — Describes the icons representing objects in the Stateflow Explorer.
- "Objects and Properties in the Contents Pane" on page 10-5 — Describes the objects and which of their properties are displayed in the Stateflow Explorer.
- "Targets in the Explorer" on page 10-7 — Describes target objects as they appear in the Stateflow Explorer.

You can also add events, data, and targets using the graphics editor **Add** menu. (See "Defining Events" on page 6-2 for more information.) However, if you add data or events via the **Add** menu, the chart is automatically defined as the parent. If you want to change the parent of a data or event object, you must use the Explorer to do so. Similarly, you must use the Explorer if you want to delete an event, data, or target object.

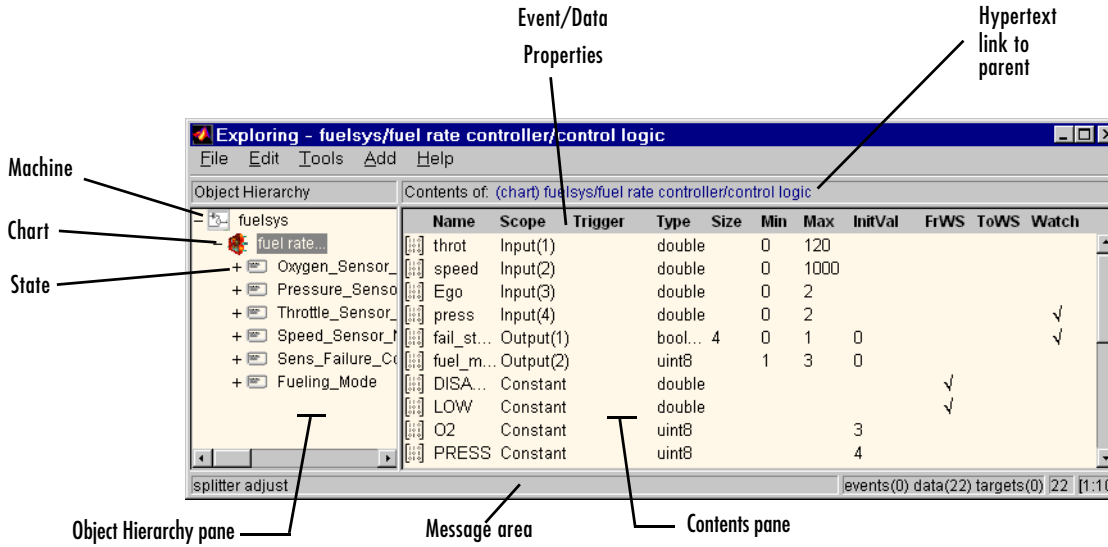
Opening Stateflow Explorer

Use one of the following methods for opening Stateflow Explorer:

- From the **Tools** menu of a Stateflow chart, select **Explore**.
- Right-click empty space in an existing Stateflow chart. In the resulting menu, select **Explore**.

Explorer Main Window

Shown below is the Explorer main window for the chart **control logic** in the demo Stateflow model `fuelsys` (Sensor Failure Detection).



The Explorer main window has two panes: an **Object Hierarchy** pane on the left and a **Contents** pane on the right. The purpose of the Explorer tool is to display the data, event, and target objects parented (contained by) each graphical element in a Stateflow chart.

The **Object Hierarchy** pane (see "Objects in the Explorer" on page 10-5) displays the graphical elements of a Stateflow chart (machines, charts, states, boxes, and graphical functions). A preceding plus (+) character indicates that you can expand the hierarchy by double-clicking that entry or by clicking the plus (+) character. A preceding minus (-) character indicates that there is nothing to expand. Clicking an entry in the **Object Hierarchy** pane selects that entry.

The **Contents** pane displays the properties of data, event, and target objects parented by the currently selected object in the **Object Hierarchy** pane. Data and event objects are described in Chapter 6, "Defining Events and Data." See "Targets in the Explorer" on page 10-7 for a quick introduction to target

objects. For a more detailed description of target objects, see Chapter 11, “Building Targets.”

Objects in the Explorer

Each type of object, whether in the **Object Hierarchy** or **Contents** pane, is displayed with an adjacent icon. In addition, objects that are subcharted (states, boxes, and graphic functions) have their appearance altered by shading. These icons also appear for matched objects in the Search & Replace tool and are as follows.

Object	Icon	Icon for Subcharted Object
Machine		Not applicable
Chart		Not applicable
State		
Box		
Graphical Function		
Data		Not applicable
Event		Not applicable
Target		Not applicable

Objects and Properties in the Contents Pane

The possible parents for the objects displayed in the **Contents** pane are shown in the following table.

Object	Can be parented by...				
	Machine?	Chart?	State?	Box?	Graphic Function?
Event	Yes	Yes	Yes	Yes	Yes
Data	Yes	Yes	Yes	Yes	Yes
Target	Yes				

For convenience, a hypertext link to the parent of the currently selected object in the **Object Hierarchy** is included following the **Contents of** label at the top of the **Contents** pane. Click this link to display that object in the Stateflow diagram editor.

Properties for each of the objects in the **Contents** pane are displayed in column entries as follows.

Property	Description	Object
`Name`	Name of object	Data, Event, Target
`Scope`	Scope of object	Data, Event
`Trigger`	Trigger of event	Event
`Type`	Data type	Data
`Size`	Size of data array	Data
`Min`	Minimum value of data limit range	Data
`Max`	Maximum value of data limit range	Data
`InitVal`	Initial value of data when initialized from data dictionary	Data
`FrWS`	Initialize data from MATLAB Workspace	Data

Property	Description	Object
`ToWS`	Save data to MATLAB workspace	Data
`Watch`	Watch in debugger	Data

The preceding properties are described in the following sections:

- "Setting Data Properties" on page 6-16
- "Setting Event Properties" on page 6-4
- "Accessing the Target Builder Dialog for a Target" on page 11-9

You can access the **Properties** dialog of each of the preceding objects to change or add properties. You can also change these properties directly from the Explorer window. See "Setting Properties for Data, Events, and Targets" on page 10-9.

Targets in the Explorer

Targets are parented exclusively by machines. (Although all other combinations are valid, there are guidelines describing how scope affects choice of parent and the reverse.) The default simulation target (`sfun`) is automatically defined for every machine. If you have a Real-Time Workshop license, a Real-Time Workshop target (`rtw`) is also automatically added when you do the following:

- Select **Open RTW Target** from the graphics editor **Tools** menu
- Build a target that includes a Stateflow machine, using Real-Time Workshop

See "Configuring a Target" on page 11-7 for information on configuring and customizing a target. See "Adding a Target to a Stateflow Machine" on page 11-7 for information on creating targets to generate code using the Stateflow Coder product.

Stateflow Explorer Operations

The Stateflow Explorer is host to a large number of operations you can perform on Stateflow objects. Many are the same operations that you can perform in the Stateflow diagram editor. However, the Explorer plays host to operations involving nongraphical objects (data, events, targets), which are available only in the Explorer.

Stateflow Explorer operations redundant to the Stateflow diagram editor are described in the following sections:

- "Opening a New or Existing Simulink Model" on page 10-9 — Shows you how to open a new or existing Simulink model from the Stateflow Explorer.
- "Editing States or Charts" on page 10-9 — Shows you how to edit states and charts from the Stateflow Explorer.
- "Setting Properties for Data, Events, and Targets" on page 10-9 — Shows you how to set the properties of data, events, and targets from the Stateflow Explorer.

The operations unique to the Stateflow Explorer are described in the following sections:

- "Moving and Copying Events, Data, and Targets" on page 10-10 — Shows you how to move and copy events, data, and targets to different owning objects in Stateflow Explorer.
- "Changing the Index Order of Inputs and Outputs" on page 10-11 — Shows you how to change the order of input and output data and event ports as they appear on the Stateflow block in the Simulink model.
- "Deleting Events, Data, and Targets" on page 10-12 — Shows you how to delete events, data, and targets in the Explorer
- "Setting Properties for the Stateflow Machine" on page 10-12 — Shows you how to set properties for a Stateflow machine, the collection of all Stateflow blocks in a Simulink model, from the Stateflow Explorer.
- "Transferring Properties Between Objects" on page 10-14 — Shows you how to transfer the properties of one data, event, or target to another object of the same type.

Opening a New or Existing Simulink Model

You can open a new Simulink model from the Stateflow Explorer by selecting **New Model** from the **File** menu. Simulink opens a new model with a default Stateflow block, which is displayed in the Explorer.

You can open an existing Simulink model from the Stateflow Explorer as follows:

1. From the **File** menu select **Open Model**.
2. In the resulting **Select File to Open** window, browse for the file to open and select **Open**.

   Simulink opens an existing model. If it includes Stateflow blocks, they are displayed in the Explorer.

Editing States or Charts

To edit a state or chart displayed in the Explorer's **Object Hierarchy** pane, do the following:

1. Right-click the object.
2. Select **Edit** from the resulting menu.

   Stateflow displays the selected object highlighted in the Stateflow editor in the context of its parent.

Setting Properties for Data, Events, and Targets

To set properties for an object displayed in the **Object Hierarchy** or **Contents** pane of Explorer, do the following:

1. Right-click the object.
2. Select **Properties** from the resulting menu.

   The properties dialog for the object appears.
3. Edit the appropriate properties and select **Apply** or **OK** to apply the changes.

You can also change properties in the Explorer directly by first clicking the object's row to highlight it and then doing the following:

- To change the **Name** property, double-click it and edit it in the resulting overlay edit field. Press the **Return** key or click anywhere outside the edit field to apply the changes.
- To change the **Scope**, **Trigger**, or **Type** properties, click the item and select from the resulting submenu.
- To change all other properties but the selected ones, click the property and edit in the resulting overlay edit field. When finished, press the **Return** key or click anywhere outside the edit field to apply the changes.
- Click a checked property to clear it. Click an unchecked property to select it.

For a list of properties you can change in the Explorer, see "Objects and Properties in the Contents Pane" on page 10-5.

Moving and Copying Events, Data, and Targets

Note If you move an object to a level in the hierarchy that does not support that object's current **Scope** property, the **Scope** is automatically changed to **Local**.

You can move event, data, or target objects to another parent by doing the following:

1. Select the events, data, and targets to move in the **Contents** pane of the Explorer.

   You can select a contiguous block of items by highlighting the first (or last) item in the block and then using **shift**+click for highlighting the last (or first) item.

2. Click and drag an object from the **Contents** pane to a new location in the **Object Hierarchy** pane to change its parent.

   A shadow copy of the selected objects accompanies the mouse cursor during dragging. If no parent is chosen or the parent chosen is the current parent,

the mouse cursor changes to an X enclosed in a circle, indicating an invalid choice.

You can accomplish the same outcome by cutting or copying the selected events, data, and targets as follows:

1. Select the event, data, and targets to move in the **Contents** pane of the Explorer.

2. From the **Edit** menu of the Explorer, select **Edit** -> **Cut** or **Copy**.

   If you select **Cut**, the selected items are deleted and are copied to the clipboard for copying elsewhere. If you select **Copy**, the selected items are left unchanged.

   You can also right-click a single selection and select **Cut** or **Copy** from the resulting menu. Explorer also uses the keyboard equivalents of **Ctrl+X** (Cut) and **Ctrl+V** (Paste).

3. Select a new parent machine, chart, or state in the **Object Hierarchy** pane of the Explorer.

4. From the **Edit** menu of the Explorer, select **Edit** -> **Paste**. The cut items appear in the **Contents** pane of the Explorer.

   You can also paste the cut items by right-clicking an empty part of the **Contents** pane of the Explorer and selecting **Paste** from the resulting menu.

Changing the Index Order of Inputs and Outputs

You can change the order of indexing for event and/or data objects with a scope of **Input(1,2,...,m)** or **Output(1,2,...,n)** to Simulink in the **Contents** pane of Explorer as follows:

1. Select one of the input or output items.

2. Drag the item to a new location within its respective list.

   A separate list can appear for input and output data, and input and output events. You must keep the dragged item within its list. Valid new locations appear with an arrow and attached text `"insert here"`.

Deleting Events, Data, and Targets

Delete event, data, and target objects in the **Contents** pane of the Explorer as follows:

1 Select the object.

2 Press the **Delete** key.

You can also select **Cut** from the **Edit** menu or **Ctrl+X** from the keyboard to delete an object.

Setting Properties for the Stateflow Machine

The Stateflow machine is Stateflow's interface to the Simulink model containing Stateflow Chart blocks (that is, charts). To set properties for the Stateflow machine, do the following:

1 Right-click the top-level machine .

2 From the resulting menu, select properties.

The **Machine properties** dialog box appears.

3 Enter information in the fields provided.

4 Click one of the following buttons:

- **Apply** saves the changes.
- **Cancel** closes the dialog without making any changes.
- **OK** saves the changes and closes the dialog box.
- **Help** displays the Stateflow online help in an HTML browser window.

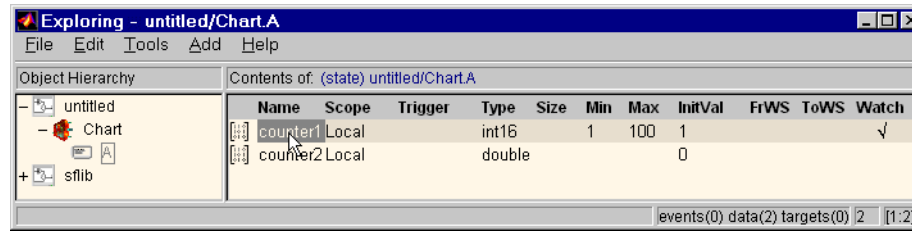
The following table lists and describes the fields of the **Machine properties** dialog.

Field	Description
Simulink Model	Name of the Simulink model that defines this Stateflow machine, which is read-only. You change the model name in Simulink when you save the model under a chosen file name.
Creation Date	Date on which this machine was created.
Creator	Name of the person who created this Stateflow machine.
Modified	Time of the most recent modification of this Stateflow machine.
Version	Version number of this Stateflow machine.
Enable C-like bit operations	If you select this box, all new charts recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode these operators as C bitwise operations. You can enable or disable this option for individual charts in the chart's property dialog box. See “Specifying Chart Properties” on page 5-78 for a detailed explanation of this property.
Description	Brief description of this Stateflow machine, which is stored with the model that defines it.
Document Link	MATLAB expression that, when evaluated, displays documentation for this Stateflow machine.

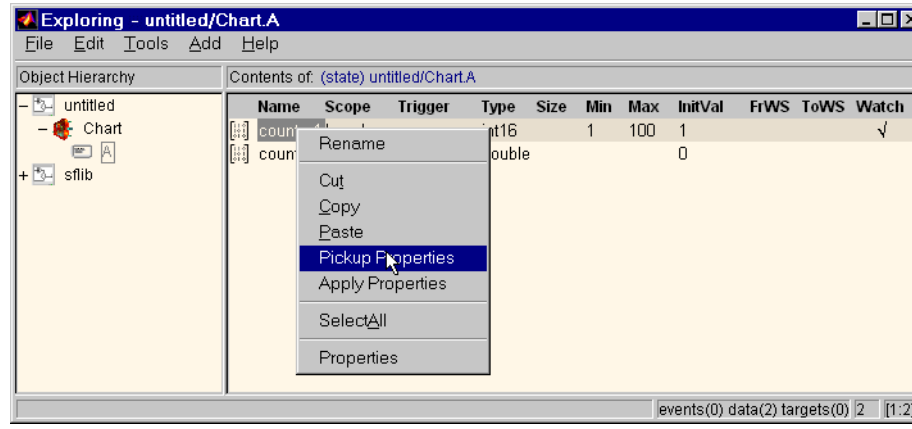
Transferring Properties Between Objects

The Explorer allows you to transfer the properties of one object to another object or set of objects with the following procedure:

1 Right-click the object in the **Contents** pane of the Explorer.

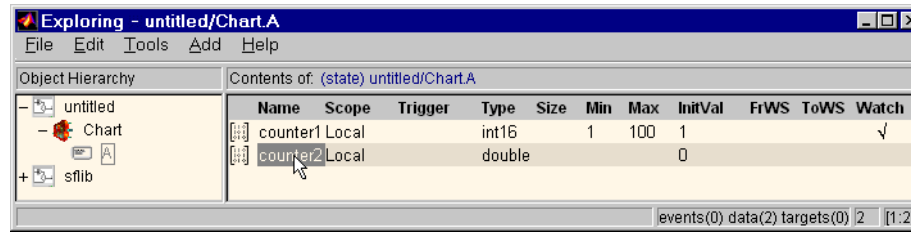

Exploring - untitled/Chart.A
File Edit Tools Add Help
Object Hierarchy
Contents of: (state) untitled/Chart.A
untitled
Chart
sflib
Name Scope Trigger Type Size Min Max InitVal FrWS ToWS Watch
counter1 Local int16 1 100 1 √
counter2 Local double 0
events(0) data(2) targets(0) 2 [1:2]

2 From the resulting menu, select **Pickup Properties**.



As an alternative to steps 1 and 2, you can also highlight the object and select **Pickup Properties** from the **Edit** menu.

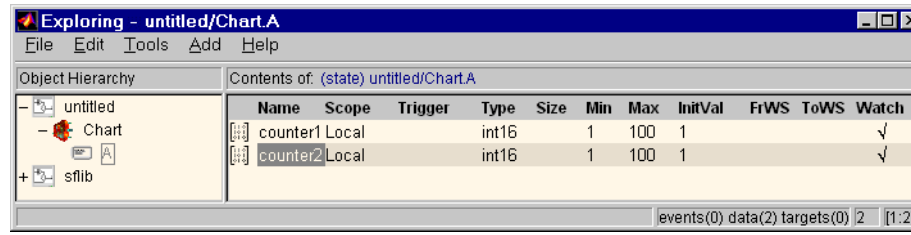
3 Select the object or objects to which you want to transfer the properties.



4 From the **Edit** menu select **Apply Properties**.

As an alternative to steps 3 and 4, if only one object is selected to receive properties, you can right-click it and select **Apply Properties** from the resulting menu.

Stateflow applies the copied properties to the selected objects, as shown in the **Type**, **Min**, **Max**, **InitVal**, and **Watch** columns of the following example:


Exploring - untitled/Chart.A
File Edit Tools Add Help
Object Hierarchy
Contents of: (state) untitled/Chart.A
untitled
Chart
A
sflib
Name Scope Trigger Type Size Min Max InitVal FrWS ToWS Watch
counter1 Local int16 1 100 1 √
counter2 Local int16 1 100 1 √
events(0) data(2) targets(0) 2 [1:2]

The Stateflow Search & Replace Tool

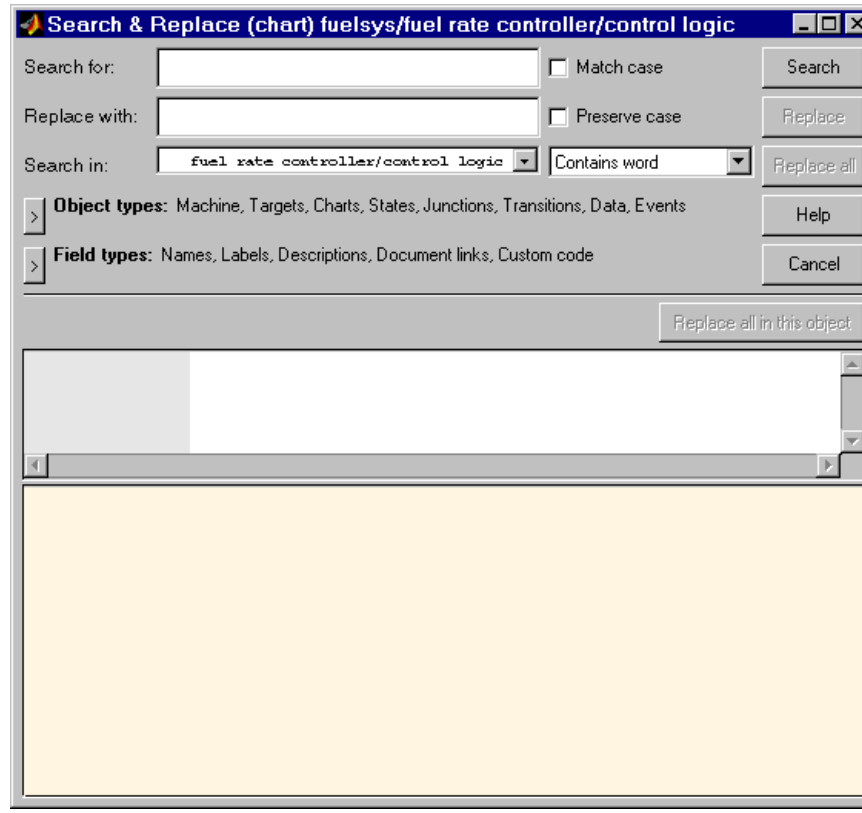
Based on textual criteria that you specify, the Stateflow Search & Replace tool searches for and replaces text belonging to objects in Stateflow charts. The following topics describe the Stateflow Search & Replace tool:

- "Opening the Search & Replace Tool" on page 10-16 — Tells you how to open the Search & Replace tool and describes the parts of the resulting dialog.
- "Using Different Search Types" on page 10-19 — Describes the different types of text searches available in the Search & Replace tool.
- "Specify the Search Scope" on page 10-21 — Tells you how to specify the parts of the model that you want to search.
- "Using the Search Button and View Area" on page 10-23 — Tells you how to use the Search button and how to interpret the display of found objects.
- "Specifying the Replacement Text" on page 10-26 — Tells you what you can specify for replacement text for the objects found by text.
- "Using the Replace Buttons" on page 10-27 — Interprets the replace buttons available and describes their use in text replacement.
- "Search and Replace Messages" on page 10-28 — Describes the text and defining icon for the informational and warning messages that appear in the Full Path Name Containing Object field.
- "The Search & Replace Tool Is a Product of Stateflow" on page 10-30 — Shows you how the Search & Replace tool is designed and implemented in Stateflow.

Opening the Search & Replace Tool

To display the **Search & Replace** dialog box, do the following:

1. Open a Stateflow chart in the Stateflow chart editor.
2. Select **Search & Replace** from the Stateflow Editor's **Tools** menu.



The window name for the **Search & Replace** dialog box contains a full path expression for the current Stateflow chart or machine in the following form.

```
(object) Machine/Subsystem/Chart
```

The **Search & Replace** dialog box contains the following fields:

- **Search for**

  Enter search pattern text in the **Search for** text box. Interpretation of the search pattern is selected with the **Match case** check box and the **Match Options** field.

- **Match case**

  If this check box is selected, the search is case sensitive and the Search & Replace tool finds only text matching the search pattern exactly. See "Match case (Case Sensitive)" on page 10-19.

- **Replace with**

  Specify the text to replace the text found when you select any of the **Replace** buttons (**Replace**, **Replace All**, **Replace All in This Object**). See "Using the Replace Buttons" on page 10-27.

- **Preserve case**

  This option modifies replacement text. For an understanding of this option, see "Replacing with Case Preservation" on page 10-26.

- **Search in**

  By default, the Search & Replace tool searches for and replaces text only within the current Stateflow chart that you are editing in the Stateflow chart editor. You can select to search the machine owning the current Stateflow chart or any other loaded machine or chart by accessing this selection box.

- **Match options**

  This field is unlabeled and just to the right of the **Search in** field. You can modify the meaning of your search text by entering one of the selectable search options. See "Using Different Search Types" on page 10-19.

- **Object types** and **Field types**

  Under the **Search in** field are the selection boxes for **Object types** and **Field types**. These selections further refine your search and are described below. By default, these boxes are hidden; only current selections are displayed next to their titles.

  Select the right-facing arrow button in front of the title to expand a selection box and make changes.

  Select the same button (this time with a left-facing arrow) to compress the selection box to display the settings only, or, if you want, just leave the box expanded.

- **Search** and **Replace** buttons

  These are described in "Using the Search Button and View Area" on page 10-23 and "Using the Replace Buttons" on page 10-27.

- **View Area**

  The bottom half of the **Search & Replace** dialog box displays the result of a search. This area is described in "A Breakdown of the View Area" on page 10-24.

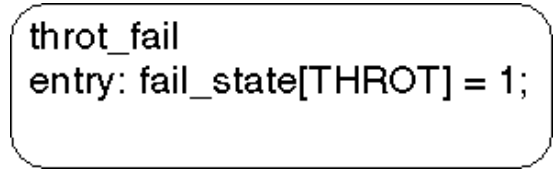
Using Different Search Types

Enter search pattern text in the **Search for** text box. You can use one of the following settings in the **Match options** field (unlabeled and just to the right of the **Search in** field) to further refine the meaning of the text entered.

Contains word

Select this option to specify that the search pattern text is a whole word expression used in a Stateflow chart with no specific beginning and end delimiters. In other words, find the specified text in any setting.

The following example is taken from the Sensor Failure Detection demo model.



Searching for the string `fail` with the **Contains word** option set finds both occurrences of the string `fail`.

Match case (Case Sensitive)

By selecting the **Match case** option, you enable case-sensitive searching. In this case, the Search & Replace tool finds only text matching the search pattern exactly.

By clearing the **Match case** option, you enable case-insensitive searching. In this case, search pattern characters entered in lower- or uppercase find matching text strings with the same sequence of base characters in lower- or uppercase. For example, the search string `"AnDrEw"` finds the matching text `"andrew"` or `"Andrew"` or `"ANDREW"`.

Match whole word

Select this option to specify that the search pattern text in the **Search for** field is a whole word expression used in a Stateflow chart with beginning and end delimiters consisting of a blank space or a character that is not alphanumeric and not an underscore character (_).

In the preceding example from the Sensor Failure Detection demo model, if **Match whole word** is selected, searching for the string `fail` finds no text within the above state. However, searching for the string `"fail_state"` does find the text `"fail_state"` as part of the second line since it is delimited as a word by a space on the front and a left square bracket (`[`) on the back.

Regular expression

Set the **Match options** field to **Regular expression** to search for text that varies from character to character within defined limits.

A regular expression is a string composed of letters, numbers, and special symbols that defines one or more string candidates. Some characters have special meaning when used in a regular expression, while other characters are interpreted as themselves. Any other character appearing in a regular expression is ordinary, unless a backslash (`\`) character precedes it.

If the **Match options** field is set to **Regular expression** in the preceding example from the Sensor Failure Detection demo model, searching for the string `"fail_"` matches the `"fail_"` string that is part of the second line, character for character. Searching with the regular expression `"\w*_"` also finds the string `"fail_"`. This search string uses the regular expression shorthand `"\w"` that represents any part-of-word character, an asterisk (`*`), which represents any number of any characters, and an underscore (_), which represents itself.

For a list of regular expression metacharacters, see the topic “Regular Expressions” in MATLAB documentation.

Searching with Regular Expression Tokens

Within a regular expression, you use parentheses to group characters or expressions. For example, the regular expression `"and(y|rew)"` matches the text `"andy"` or `"andrew"`. Parentheses also have the side effect of remembering what they match so that you can recall and reuse the found text with a special variable in the **Search for** field. These are referred to as *tokens*.

For an understanding of how to use tokens to enhance searching in the Search & Replace tool, see the topic “Searching with Tokens” in MATLAB documentation.

You can also use tokens in the **Replace with** field. See “Replacing with Tokens” on page 10-27 in for a description of using regular expression tokens for replacing.

Preserve case

This option actually modifies replacement text and not search text. For an understanding of this option, see “Replacing with Case Preservation” on page 10-26.

Specify the Search Scope

You specify the search scope for your search by selecting from the field regions discussed in the following topics:

- “Search in” on page 10-21 — Select a whole machine or individual Stateflow chart for searching.
- “Object Types” on page 10-22 — Limit your search to text matches in the selected object types.
- “Field Types” on page 10-22 — Limit your search to text matches for the specified fields

Search in

You can select a whole machine or individual Stateflow chart for searching in the **Search in** field. By default, the current Stateflow chart in which you entered the Search & Replace tool is selected.

To select a machine, do the following:

1. Select the down arrow of the **Search in** field.

   A list of the currently loaded machines appears with the current machine expanded to reveal its underlying Stateflow charts.

2. Select a machine.

To select a Stateflow chart for searching, do the following:

1 Select the down arrow of the **Search in** field again.

   This time the displayed list contains the previously selected machine expanded to reveal its Stateflow charts.

2 Select a chart from the expanded machine.

Object Types

Limit your search to text matches in the selected object types only when you do the following:

1 Expand the **Object types** field.

2 Select one or more object types.

Field Types

Limit your search to text matches for the specified fields only by doing the following:

1 Expand the **Field types** field.

2 Select one or more of the available field types

Available field types are as follows:

Names. Machines, charts, data, and events have valid **Name** fields. States have a **Name** defined as the top line of their labels. You can search and replace text belonging to the **Name** field of a state in this sense. However, if the Search & Replace tool finds matching text in a state's **Name** field, the remainder of the label is subject to succeeding searches for the specified text whether or not the label is chosen as a search target.

Note The **Name** field of machines and charts is an invalid target for the Search & Replace tool. Use Simulink to change the names of machines and charts.

Labels. Only states and transitions have labels.

Descriptions. All objects have searchable **Description** fields.

Document links. All objects have searchable **Link** fields.

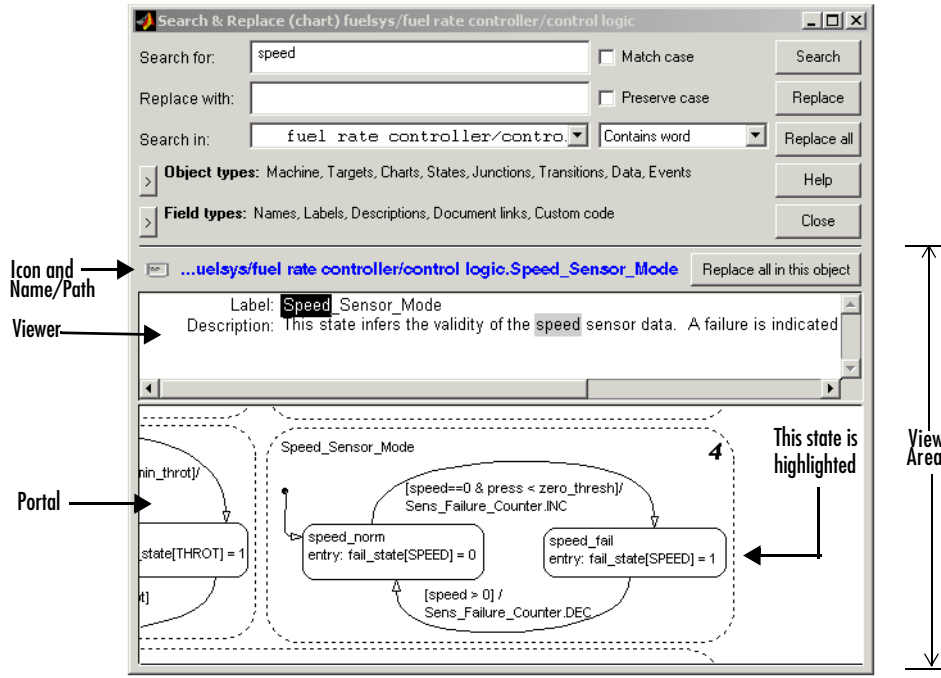
Custom code. Only target objects contain custom code.

Using the Search Button and View Area

This topic contains the following subtopics:

- "A Breakdown of the View Area" on page 10-24
- "The Search Order" on page 10-24
- "Additional Display Options" on page 10-25

Click **Search** to initiate a single-search operation. If an object match is made, its text fields are displayed in the **Viewer** pane in the middle of the **Search & Replace** dialog. If the object is graphical (state, transition, junction, chart), the matched object is displayed in a **Portal** pane below the **Viewer** pane.



A Breakdown of the View Area

The view area of the **Search & Replace** dialog box displays found text and its containing object, if viewable. In the preceding example, taken from the Sensor Fuel Detection demo model, a search for the word `"speed"` finds the **Description** field for the state `Speed_Sensor_Mode`. The resulting view area display consists of the following parts:

Icon. Displays an icon appropriate to the object containing the found text. These icons are identical to the icons used in the Stateflow Explorer and are displayed in "Explorer Main Window" on page 10-4.

Full Path Name of Containing Object. This area displays the full path name for the object containing the found text in the following format:

```
(<type>) <machine name>/<subsystem>/<chart
name>.[p1]...[pn].<object name> (<id>)
```

where p_1 through p_n denote the object's parent states.

To display the object, click the mouse once on the full path name of the object. If the object is a graphical member of a Stateflow chart, it is displayed in the Stateflow chart editor. Otherwise, it is displayed as a member of its Stateflow chart in the Stateflow Explorer.

Viewer. This area displays the found text as a highlighted part of all search-qualified text fields for the owner object. If other occurrences exist in these fields, they too are highlighted, but in lighter shades.

To invoke the **Properties** dialog box for the owner object, double-click anywhere in the view area.

Portal. This area contains a graphic display of the object containing the matching text. The object containing the found text is highlighted in blue (default).

To display the highlighted object in the Stateflow chart editor window, double-click anywhere in the portal.

The Search Order

If you specify an entire machine as your search scope in the **Search in** field, the Search & Replace tool starts searching at the beginning of the first chart of the model, regardless of the Stateflow chart displayed in the Stateflow chart editor

when you begin your search. After searching the first chart, the Search & Replace tool continues searching each chart in model order until all charts for the model have been searched.

If you specify a Stateflow chart as your search scope, the Search & Replace tool begins searching at the beginning of the chart. The Search & Replace tool continues searching the chart until all the chart's objects have been searched.

The search order taken in searching an individual chart for matching text is equivalent to a depth-first search of the Stateflow Explorer. Starting at the highest level of the chart, the Explorer hierarchy is traversed downward from parent to child until an object with no child is encountered. At this point, the hierarchy is traversed upward through objects already searched until an unsearched sibling is found and the process is repeated.

Additional Display Options

Right-click anywhere in the **Search & Replace** dialog to display a menu with the following selections.

Selection	Result
Show Portal	A toggle switch that hides or displays the portal.
Edit	Displays the object with the matching text in the Stateflow chart editor; applies to states, junctions, transitions, and charts.
Explore	Displays the object with the matching text in the Stateflow Explorer. Applies to states, data, events, machines, charts, and targets.
Properties	Displays the **Properties** dialog box for the object with the matching text.

Note The **Edit**, **Explore**, and **Properties** selections are enabled only after a successful search.

If the portal is not visible, you can select the **Show Portal** option to display it. You can also simply click-drag the border between the viewer and the portal (the cursor turns to a vertical double arrow), which resides just above the bottom boundary of the **Search & Replace** dialog. Moving this border allows you to exchange area between the portal and the viewer. If you click-drag the border with the left mouse button, the graphic display resizes after you reposition the border. If you click-drag the border with the right mouse button, the graphic display continuously resizes as you move the border.

Specifying the Replacement Text

The Search & Replace tool replaces found text with the exact (case-sensitive) text you entered in the **Replace With** field unless you choose one of the dynamic replacement options described below.

Replacing with Case Preservation

If you choose the **Case Preservation** option, matched text is replaced based on one of the following conditions discovered in the found text:

- Whisper

  In this case, the found text has no uppercase characters, only lowercase. Found text is replaced entirely with the lowercase equivalent of all replacement characters. For example, if the replacement text is `"ANDREW"`, the found text `"bill"` is replaced by `"andrew"`.

- Shout

  In this case, the found text contains only uppercase characters. Found text is replaced entirely with the uppercase equivalent of all replacement characters. For example, if the replacement text is `"Andrew"`, the found text `"BILL"` is replaced by `"ANDREW"`.

- Proper

  In this case, the found text contains uppercase characters in the first character position of each word. Found text is replaced entirely with the case equivalent of all replacement characters. For example, if the replacement text is `"andrew johnson"`, the found text `"Bill Monroe"` is replaced by `"Andrew Johnson"`.

- Sentence

  In this case, the found text contains an uppercase character in the first character position of a sentence with all remaining sentence characters in lowercase. Found text is replaced in like manner, with the first character of the sentence given an uppercase equivalent and all remaining sentence characters set to lowercase. For example, if the replacement text is `"andrew is tall."`, the found text `"Bill is tall."` is replaced by `"Andrew is tall."`.

Replacing with Tokens

Within a regular expression, you use parentheses to group characters or expressions. For example, the regular expression `"and(y|rew)"` matches the text `"andy"` or `"andrew"`. Parentheses also have the side effect of remembering what they matched so that you can recall and reuse the found text with a special variable in the **Replace with** field. These are referred to as *tokens*.

Tokens outside the search pattern have the form `$1,$2,...,$n (n<17)` and are assigned left to right from parenthetical expressions in the search string.

For example, the search pattern `"(\w*)_(\w*)"` finds all word expressions with a single underscore separating the left and right sides of the word. If you specify an accompanying replacement string of `"$2_$1"`, you can replace all these expressions by their reverse expression with a single **Replace all**. For example, the expression `"Bill_Jones"` is replaced by `"Jones_Bill" and` the expression `"fuel_system"` is replaced by `"system_fuel".`

For a clearer understanding of how tokens are used in regular expression search patterns, see “Regular Expressions” in MATLAB documentation.

Using the Replace Buttons

You can activate the replace buttons (**Replace**, **Replace All**, **Replace All in This Object**) only after a search that finds text.

Replace

When you select the **Replace** button, the current instance of text matching the text string in the **Search for** field is replaced by the text string entered in the **Replace with** field. The Search & Replace tool then automatically searches for the next occurrence of the **Search for** text string.

Replace All

When you select the **Replace All** button, all instances of text matching the **Search for** field are replaced by the text string entered in the **Replace with** field. Replacement starts at the point of invocation to the end of the current Stateflow chart. This means that if you initially skip through some search matches with the **Search** button, they are also skipped when you select the **Replace All** button.

If the search scope is set to **Search Whole Machine**, then after finishing the current Stateflow chart, replacement continues to the completion of all remaining charts in your Simulink model.

Replace All in This Object

When you select the **Replace All in This Object** button, all instances of text matching the **Search for** field are replaced by text entered in the **Replace with** field everywhere in the current Stateflow object regardless of previous searches.

Search and Replace Messages

Informational and warning messages appear in the **Full Path Name Containing Object** field along with a defining icon.

- Informational Messages

- Warnings

The following messages are informational only:

Please specify a search string

A search was attempted without a search string specified.

No Matches Found

There are no matches within the selected search scope.

Search Completed

There are no more matches within the selected search scope.

The following messages are warnings that refer to invalid conditions for searching or replacing:

Invalid option set

The object types and field types that you have selected are incompatible. For example, a search on **Custom Code** fields without selecting target objects is invalid.

Match object not currently editable

The found object is not editable by replacement because of one of the following.

Problem	Solution
A simulation is running.	Stop the simulation.
You are editing a locked library block.	Unlock the library.
The current object or its parent has been manually locked.	Unlock the object or its parent.

The following messages are warnings that, when the Search & Replace tool performs a search or replacement immediately after finding an object, it must first refind the object and its matching text field. If that original found object is deleted or changed before an ensuing search or replacement, the Search & Replace tool cannot continue:

Search object not found

If you search for text, find it, and then delete the containing object, this warning results if you continue to search.

Match object not found

If you search for text, find it, and then delete the containing object, this warning results if you perform a replacement.

Match not found

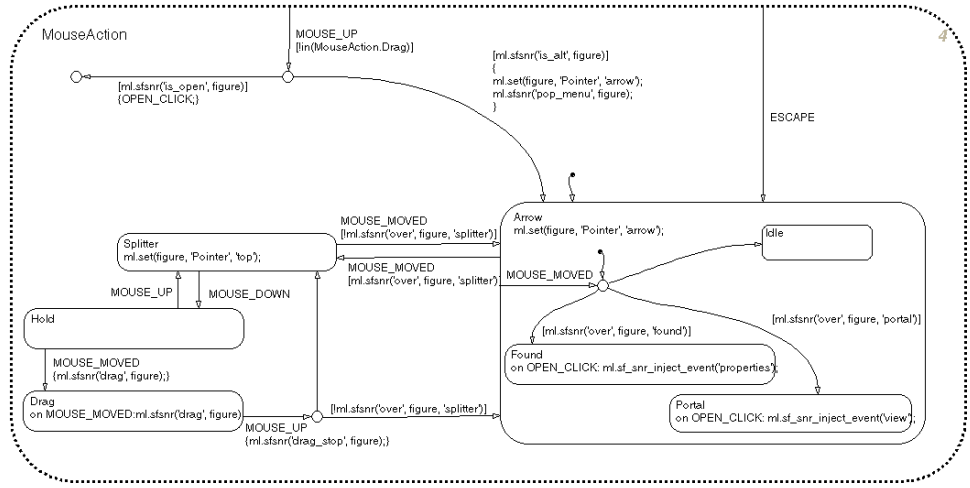
If you search for text, find it, and then change the object containing the text, this warning results if you perform a replacement.

Search string changed

If you search for text, find it, and then change the **Search For** field, this warning results if you perform a replacement.

The Search & Replace Tool Is a Product of Stateflow

The Search & Replace tool in Stateflow is a product of Stateflow itself. The following Stateflow chart implements the behavior of the mouse in the Search & Replace tool.



The Stateflow Finder Tool

There are two varieties of tools that search only for Stateflow, depending on your platform. These are as follows:

- The Simulink Find tool allows you to search Stateflow models for Simulink and Stateflow objects, such as states and transitions, that meet criteria you specify. Simulink displays any objects that satisfy the search criteria in the dialog box's search results pane. See "Searching for Objects" in the Using Simulink documentation for information on using the Simulink **Find** dialog box.
- On platforms that do not support the Simulink Find tool, the original Stateflow Finder appears when you select **Find** from the Stateflow Editor's **Tools** menu. The following topics explain how to use the original Stateflow Finder to search for objects.
  - "Opening Stateflow Finder" on page 10-31 — Tells you how to open the Stateflow Finder tool.
  - "Using Stateflow Finder" on page 10-32 — Describes the fields and selections available for describing the objects you want to find.
  - "Finder Display Area" on page 10-35 — Describes the display columns for found items in the display area.

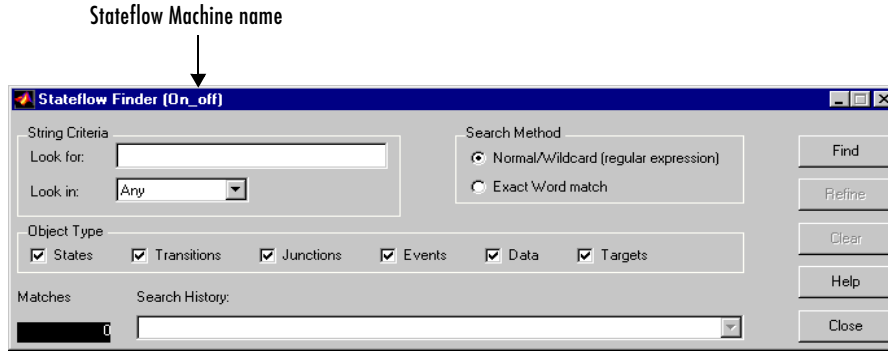
Note See the Simulink Release Notes in the online documentation for a list of platforms on which the Simulink Find tool is not available.

Opening Stateflow Finder

On platforms that do not support the Simulink Find tool (see preceding note), display the Stateflow **Finder** dialog box with one of the following:

- Select **Find** from the Stateflow Editor's **Tools** menu.
- Select **Find** from the Simulink model window's **Edit** menu.

The Finder operates on the machine whose name appears in the window title area of the Finder dialog as shown:



Using Stateflow Finder

The following topics in this section describe the parts of the Stateflow Finder:

- "String Criteria" on page 10-32
- "Search Method" on page 10-33
- "Object Type" on page 10-34
- "Find Button" on page 10-34
- "Matches" on page 10-34
- "Refine Button" on page 10-34
- "Search History" on page 10-34
- "Clear Button" on page 10-35
- "Close Button" on page 10-35
- "Help Button" on page 10-35

String Criteria

You specify the string by entering the text to search for in the **Look for** text box. The search is case sensitive. All text fields are included in the search by default. Alternatively, you can search in specific text fields by using the **Look in** list box to choose one of these options:

Any. Search the state and transition labels, object names, and descriptions of the specified object types for the string specified in the **Look for** field.

Label. Search the state and transition labels of the specified object types for the string specified in the **Look for** field.

Name. Search the **Name** fields of the specified object types for the string specified in the **Look for** field.

Description. Search the **Description** fields of the specified object types for the string specified in the **Look for** field.

Document Link. Search the **Document** link fields of the specified object types for the string specified in the **Look for** field.

Custom Code. Search custom code for the string specified in the **Look for** field.

Search Method

By default the **Search Method** is **Normal/Wildcard** (regular expression). Alternatively, you can click the **Exact Word match** option if you are searching for a particular sequence of one or more words.

A regular expression is a string composed of letters, numbers, and special symbols that define one or more strings. Some characters have special meaning when used in a regular expression, while other characters are interpreted as themselves. Any other character appearing in a regular expression is ordinary, unless a \ precedes it.

These are the special characters supported by Stateflow Finder.

Character	Description
^	Start of string
$	End of string
.	Any character
\	Quote the next character
*	Match zero or more
+	Match one or more
[ ]	Set of characters

Object Type

Specify the object types to search by toggling the check boxes. A check mark indicates that the object is included in the search criteria. By default, all object types are included in the search criteria. **Object Types** include the following:

- States
- Transitions
- Junctions
- Events
- Data
- Targets

Find Button

Click the **Find** button to initiate the search operation. The data dictionary is queried and the results are listed in the display area.

Matches

The **Matches** field displays the number of objects that match the specified search criteria.

Refine Button

After the results of a search are displayed, enter additional search criteria and click **Refine** to narrow the previously entered search criteria. An ampersand (&) is prefixed to the search criteria in the **Search History** field to indicate a logical AND with any previously specified search criteria.

Search History

The **Search History** text box displays the current search criteria. Click the pull-down list to display search refinements. An ampersand is prefixed to the search criteria to indicate a logical AND with any previously specified search criteria. You can undo a previously specified search refinement by selecting a previous entry in the search history. By changing the **Search History** selection you force the Finder to use the specified criteria as the current, most refined, search output.

Clear Button

Click **Clear** to clear any previously specified search criteria. Results are removed and the search criteria are reset to the default settings.

Close Button

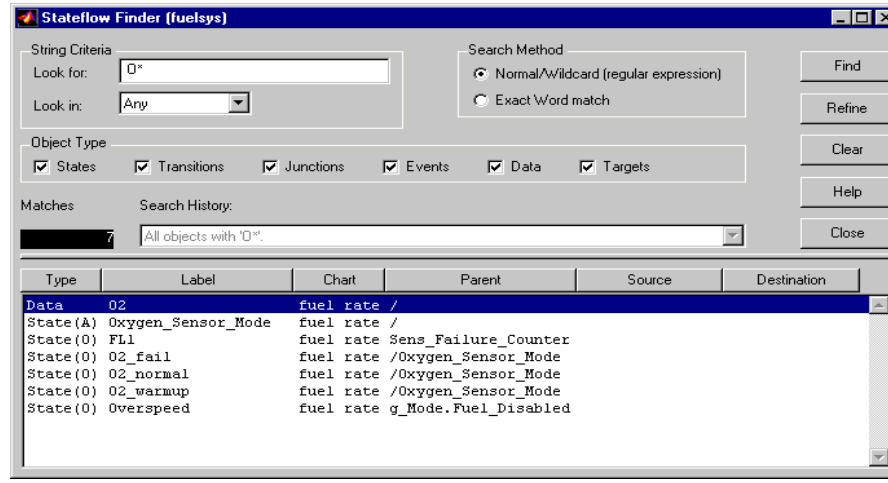
Click **Close** to close the Finder.

Help Button

Click **Help** to display the Stateflow online help in an HTML browser window.

Finder Display Area

The Finder display area has an appearance similar to the following.



The display area displays found entries with the following columns:

Field	Description
Type	The object type is listed in this field. States with exclusive (OR) decomposition are followed by an (O). States with parallel (AND) decomposition are followed by (A).
Label	The string label of the object is listed in this field.
Chart	The title of the Stateflow diagram (Stateflow block) is listed in this field.
Parent	This object's parent in the hierarchy.
Source	Source object of a transition.
Destination	Destination object of a transition.

All fields are truncated to maintain column widths. The **Parent**, **Source**, and **Destination** fields are truncated from the left so that the name at the end of the hierarchy is readable. The entire field contents, including the truncated portion, are used for resorting.

Each field label is also a button. Click the button to have the list sorted based on that field. If the same button is pressed twice in a row, the sort ordering is reversed.

You can resize the Finder vertically to display more output rows, but you cannot expand it horizontally.

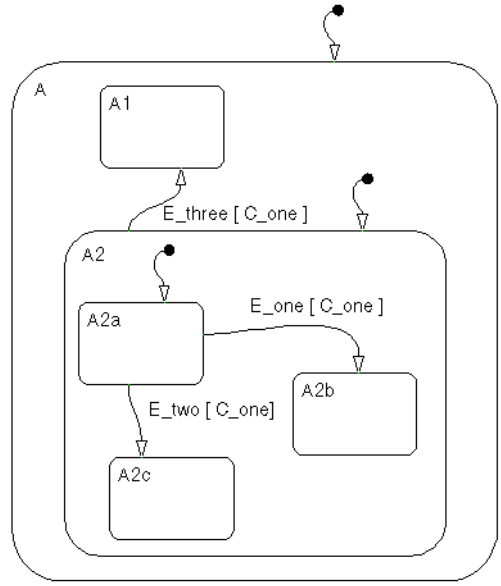
Click a graphical entry to highlight that object in the graphical editor window. Double-click an entry to invoke the **Properties** dialog box for that object. Right-click the entry to display a menu that allows you to explore, edit, or display the properties of that entry.

Representing Hierarchy

The Finder displays **Parent**, **Source**, and **Destination** fields to represent the hierarchy. The Stateflow diagram is the root of the hierarchy and is represented by the / character. Each level in the hierarchy is delimited by a period (.) character. The **Source** and **Destination** fields use the combination of

the tilde (~) and the period (.) characters to denote that the state listed is relative to the **Parent** hierarchy.

Using the following Stateflow diagram as an example, what are the values for the **Parent**, **Source**, and **Destination** fields for the transition from `A2a` to `A2b`?



The `A2a` to `A2b` transition is within state `A2`. State `A2`'s parent is state `A` and state `A`'s parent is the Stateflow diagram itself. The notation for state `A2a`'s parent is `/A.A2`. State `A2a` is the transition source and state `A2b` is the destination. These states are at the same level in the hierarchy. The relative hierarchy notation for the source of the transition is `~.A2a`. The full path is `/A.A2.A2a`. The relative hierarchy notation for the destination of the transition is `~.A2b`. The full path is `/A.A2.A2b`.

11

Building Targets

Overview of Stateflow Targets (p. 11-3)	A target is a program that executes a Stateflow model or a Simulink model containing a Stateflow machine. This section describes how Stateflow and its companion tools can build targets for virtually any computer.
Setting Up the Target Compiler (p. 11-6)	Tells you how to set up a target for your platform.
Configuring a Target (p. 11-7)	Describes the details of each step required for configuring a target.
Integrating Custom Code with Stateflow Diagrams (p. 11-20)	Tells you how to specify options for the custom code that you include in the target you build. Also tells you how to invoke Stateflow graphical functions from your custom code.
Starting the Build (p. 11-25)	Describes how to start building a simulation target (`sfun`) or a Real-Time Workshop target (`rtw`).
Parsing Stateflow Diagrams (p. 11-27)	The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. This section describes the Stateflow parser and how its messages appear.
Resolving Event, Data, and Function Symbols (p. 11-32)	Describes the process Stateflow uses when attempting to resolve undefined data, event, and graphical function symbols when you choose to simulate the model, build a target, or generate code.

Error Messages (p. 11-34)	Describes the error messages you might receive when you build a target or parse a diagram.
Generated Files (p. 11-37)	Describes the files that Stateflow generates when you generate code for a target or build the target.

Overview of Stateflow Targets

A target is a program that executes a Stateflow model or a Simulink model containing a Stateflow machine. Stateflow and companion tools can build targets for virtually any computer. The following sections describe targets in Stateflow:

- "Target Types" on page 11-3 — Describes the type of targets you can build from Stateflow.
- "Building a Target" on page 11-3 — Gives the high-level steps for building a target in Stateflow into an executable.
- "How Stateflow Builds Targets" on page 11-5 — Gives the steps that Stateflow takes in building its own targets.

Target Types

Simulink and its companion tools can build the following types of targets:

- Simulation target

  The simulation target (named `sfun`) is a compiled Simulink S-function (MEX-file) that enables Simulink to simulate a Stateflow model. See "Parsing Stateflow Diagrams" on page 11-27 for more information.

- Real-Time Workshop (RTW) target

  The RTW target (named `rtw`) is an executable program that implements a Simulink model. The model represented by an RTW target can include non-Stateflow as well as Stateflow blocks. An RTW target can also run on computers that do not have a floating-point instruction set. Building an RTW target requires the Real-Time Workshop and Stateflow Coder.

- Custom target

  Users can generate code for custom targets (any name but `sfun` or `rtw`) that they can use in building customized applications.

Building a Target

Building a target involves the following steps:

1 Configure the target.

   See "Configuring a Target" on page 11-7 for more information. You need to perform this step only if you are building a stand-alone or RTW target or are including custom code in the target. See "Building Custom Code into the Target" on page 11-4.

2 Start the build process.

   Stateflow automatically builds or rebuilds simulation targets when you initiate simulation of the Stateflow machine. You must explicitly initiate the build process for other types of targets. See "Configuring a Target" on page 11-7 for more information.

Configuring and building a target requires a basic understanding of how Stateflow builds targets, in the case of simulation and stand-alone targets, and how Real-Time Workshop builds targets, in the case of RTW targets. See "How Stateflow Builds Targets" on page 11-5 for information on how Stateflow builds targets. Real-Time Workshop uses the same basic process for building Stateflow targets that it uses for building non-Stateflow targets. See the Real-Time Workshop documentation for information on how Real-Time Workshop builds targets.

Rebuilding a Target

You can rebuild a target at any time by repeating step 2. When rebuilding a target, Stateflow rebuilds only those parts corresponding to charts that have changed logically since the last build. When rebuilding a target, you need to perform step 1 only if you want to change the target's custom code or configuration.

Building Custom Code into the Target

You can configure the target build process to include custom C code supplied by you in the target (see "Integrating Custom Code with Stateflow Diagrams" on page 11-20). You use Stateflow or Real-Time Workshop to build an entire application that includes both the portions that you supply and the target code generated by Stateflow. You use Real-Time Workshop and Stateflow when you are building applications that include other types of Simulink blocks.

How Stateflow Builds Targets

Stateflow builds a target for its diagrams as follows:

1. Stateflow parses the charts that represent the Stateflow machine to ensure that the machine's logic is valid.

2. If any errors are found, Stateflow displays the errors in the MATLAB Command Window (see "Parsing Stateflow Diagrams" on page 11-27) and halts.

3. If the charts parse without error, Stateflow next invokes a code generator to convert the charts in the Stateflow machine into C source code.

   The code generator accepts various options that control the code generation process. You can specify these options via the Stateflow user interface (see "Adding a Target to a Stateflow Machine" on page 11-7).

4. The code generator generates a makefile to build the generated source code into an executable program.

   The generated makefile can optionally build custom code that you specify into the target (see "Integrating Custom Code with Stateflow Diagrams" on page 11-20).

5. Stateflow builds the target using a C compiler and make utility that you specify (see "Setting Up the Target Compiler" on page 11-6 for more information).

Setting Up the Target Compiler

Building Simulink targets requires a C compiler that is supported by MATLAB. The Microsoft Windows version of Stateflow comes with a C compiler (`lcc.exe`) and a make utility (`lccmake`). Both tools are installed in the directory `matlabroot\sys\lcc`. If you do not configure MATLAB to use any other compiler, Stateflow uses `lcc` to build targets.

For other compilers and for all non-Windows platforms, do the following:

1. Install the C compiler you want Stateflow to use to build targets on your system.

   You can use any compiler supported by MATLAB for building MATLAB extension (MEX) files. See "Building MEX-Files" in the MATLAB External Interfaces documentation for information on C compilers supported by MATLAB.

Note Stateflow supports building targets with Microsoft Visual C/C++ 5.0 only if you have installed the Service Pack 3 updates for that product.

---

2. At the MATLAB prompt, type `mex -setup` to initiate prompts for entering the appropriate information about your compiler.

   Stateflow uses the compiler that you specify to build MEX-files for building a Stateflow simulation target.

On Windows platforms, if you want to use a compiler that you supply to build some targets and `lcc` to build other targets, first set up MATLAB to use the compiler you supply. Then select the **Use lcc compiler** option on the **Coder** dialog box (see "Simulation Target (sfun) Coder Options" on page 11-11) for each target that you want to build with `lcc`.

Configuring a Target

Configuring a target entails some or all of the steps described in the following topics:

- "Adding a Target to a Stateflow Machine" on page 11-7 — Tells you how to add a new target to the Stateflow machine's target list.
- "Accessing the Target Builder Dialog for a Target" on page 11-9 — Tells you how to start the target builder dialog box for specifying build, code generation, and custom code options.
- "Specifying Build Options for a Target" on page 11-10 — Shows you how to specify the type of build that takes place when you start the build.
- "Specifying Code Generation Options" on page 11-11 — Shows you how to specify code generation options.
- "Integrating Custom Code with Stateflow Diagrams" on page 11-20 — Tells you how to specify your own custom code to integrate with generated code in Stateflow.

For instructions on how to start the build process, see "Starting the Build" on page 11-25.

Note Configuring an RTW target might require additional steps. See the Real-Time Workshop documentation for more information.

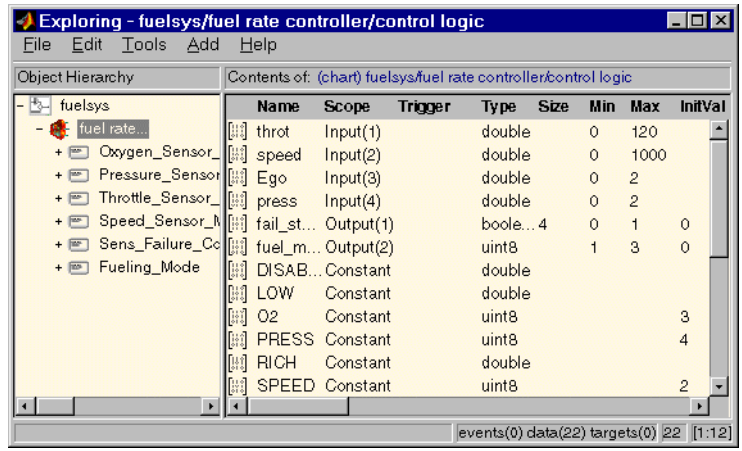
Adding a Target to a Stateflow Machine

Building a target requires that you first add the target to the list of potential targets maintained by Stateflow for a particular model.

To add a target to the model, do the following:

1 Select **Explore** from the Stateflow editor's **Tools** menu.

The Stateflow Explorer appears.



The Explorer object hierarchy shows the Stateflow machines (Simulink models) currently loaded in memory. In the preceding example, the Stateflow machine `fuelsys` is just above the chart `fuel rate controller` (shown as `fuel rate...`), which is highlighted by default.

2 Select the Stateflow machine to which you want to add the RTW target.

The Explorer displays the selected Stateflow machine's data, events, and targets in the contents pane.

3 Select **Target** from the Explorer's **Add** menu to add a target with the default name "untitled" to the selected machine.

4 Rename the target.

Follow the steps in the next topic to rename the target to `rtw`.

Note A Stateflow machine can have only one RTW target.

Renaming the Target

To rename the target:

1 Select the target in the Explorer's **Contents** pane and right-click.

2 Select **Rename** from the resulting context menu.

The Explorer redisplays the selected target's name in an edit box.

3 Change the target's name in the edit box.

4 Click outside the edit box to close it.

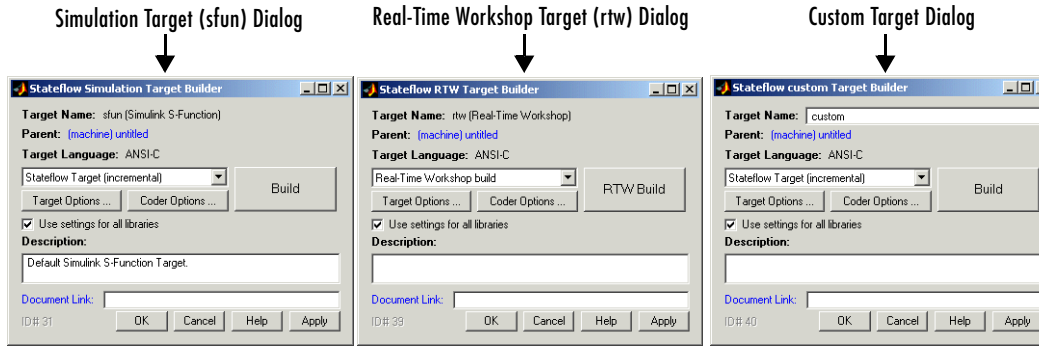
Accessing the Target Builder Dialog for a Target

To specify configuration options for any target, do the following:

1 Select the target in the Explorer's **Contents** pane and right-click.

2 Select **Properties** from the resulting context menu.

The **Target Builder** dialog box for the target's type appears as one of the following:



You can also specify the configuration options for a simulation target (`sfun`) or Real-Time Workshop target (`rtw`) by selecting one of the following from the **Tools** menu in the diagram window:

- **Open Simulation Target** to specify build options for a simulation target
- **Open RTW Target** to specify build options for an RTW target

You specify the settings for your target as described in the following topics:

- "Specifying Build Options for a Target" on page 11-10 — Build options are selected from the drop-down list left of the **Build** button.
- "Specifying Code Generation Options" on page 11-11 — Code generation options are selected through the **Coder Options** button.
- "Integrating Custom Code with Stateflow Diagrams" on page 11-20 — Custom code options are selected through the **Target Options** button.

Note If you want the settings that you specify for this target to apply to all the Stateflow charts contributed by library models as well, select (check) the **Use settings for all libraries** option.

Specifying Build Options for a Target

Select a build option for your target from the drop-down list left of the **Build** button on the **Target Builder** dialog box (see "Accessing the Target Builder Dialog for a Target" on page 11-9). This option determines the extent of the build activity that takes place when you start the build.

For the Simulation target, select from the following build options:

- **Stateflow Target (incremental**) to rebuild only those portions of the target corresponding to charts that have changed logically since the last build.
- **Rebuild All (including libraries)** to rebuild the target, including chart libraries, from scratch.
- **Make without generating code** to invoke the make process without generating code. This is useful when you have custom source files that need to be recompiled within a Stateflow incremental build mechanism that does not detect changes in custom software files.

For the RTW target, select from the following build options:

- **Real-Time Workshop build** to build or rebuild the RTW target.
- **Real-Time Workshop options** allows you to change Simulink parameters for your RTW build. Once you select this option, the button to the right of this field is renamed **RTW Options**. Selecting this button displays the **Simulation Parameters** dialog of Simulink. See "Overview of the Real-Time Workshop User Interface" in the Real-Time Workshop documentation.

- **Stateflow Target** to build only Stateflow code in the RTW target.

For a custom target, select from the following build options:

- **Stateflow Target (incremental**) to rebuild only those portions of the target corresponding to charts that have changed logically since the last build.
- **Rebuild All (including libraries)** to rebuild the target, including chart libraries, from scratch.
- **Make without generating code** to invoke the Make process without generating code. This is useful when you have custom source files that need to be recompiled within a Stateflow incremental build mechanism that does not detect changes in custom software files.
- **Generate code only (incremental)** to generate code for charts that have changed logically since the last build.
- **Generate code only (nonincremental)** to regenerate code for all charts in the model.

Specifying Code Generation Options

Specify code generation options for a target in the **Target Builder** dialog box for the target (see "Accessing the Target Builder Dialog for a Target" on page 11-9 for instructions on how to access this dialog). These options differ between targets as described in the following topics:

- "Simulation Target (sfun) Coder Options" on page 11-11
- "RTW Target (rtw) Coder Options" on page 11-13
- "Custom Target Coder Options" on page 11-17

Simulation Target (sfun) Coder Options

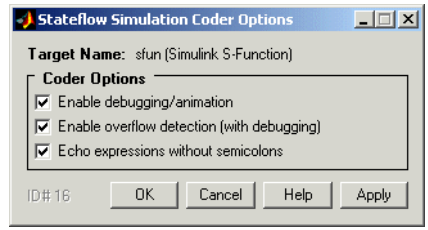
To specify code generation options for a simulation target, do the following:

1 Open the **Stateflow Simulation Target Builder** dialog.

See "Accessing the Target Builder Dialog for a Target" on page 11-9 for instructions on how to access this dialog.

2 Select **Coder Options**.

The **Simulation Coder Options** dialog box appears.



3 Select any or all of the following options:

- **Enable debugging/animation** — Enables chart animation and debugging. Stateflow enables debugging code generation when you use the debugger to start a model simulation. You can enable or disable chart animation separately in the debugger. (The Stateflow debugger does not work with stand-alone and RTW targets. Therefore, Stateflow and Real-Time Workshop do not generate debugging/animation code for these targets, even if this option is enabled.)
- **Enable overflow detection (with debugging)** — Overflow occurs for data when a value is assigned to it that exceeds the numeric capacity of its type. If this check box is selected, Stateflow generates code for overflow detection of Stateflow data. If cleared, no code is generated for overflow detection.

Note To actually detect overflow in data during simulation, you must also select the **Data Range** check box in the Debugger window. See "Debugging Data Range Violations" on page 12-20 for more details.

The **Enable overflow detection (with debugging)** option is particularly important for fixed-point data. See "Overflow Detection for Fixed-Point Types" on page 7-37.

- **Echo expressions without semicolons** — Display run-time output in the MATLAB Command Window, specifically actions that are not terminated by a semicolon.

4 Click **Apply** to apply the selected options or **OK** to apply the options and close the dialog.

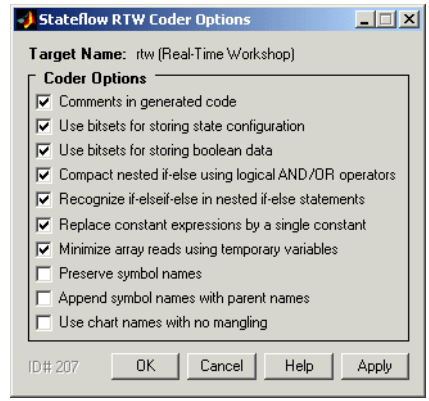
Note Use the **Chart Properties** dialog to tell the simulation target builder to recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode them as C bitwise operations.

RTW Target (rtw) Coder Options

To specify code generation options for an RTW target:

1 From the **Stateflow RTW Target Builder** dialog (see "Accessing the Target Builder Dialog for a Target" on page 11-9) select **Coder Options**.

The **RTW Coder Options** dialog box appears.


Stateflow RTW Coder Options
Target Name: rtw (Real-Time Workshop)
Coder Options
Comments in generated code
Use bitsets for storing state configuration
Use bitsets for storing boolean data
Compact nested if-else using logical AND/OR operators
Recognize if-elseif-else in nested if-else statements
Replace constant expressions by a single constant
Minimize array reads using temporary variables
Preserve symbol names
Append symbol names with parent names
Use chart names with no mangling
ID# 207
OK
Cancel
Help
Apply

2 Select any or all of the following options:

- **Comments in generated code** — Include comments in the generated code.
- **Use bitsets for storing state configuration** — Use bitsets for storing state configuration variables. This can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.
- **Use bitsets for storing boolean data** — Use bitsets for storing Boolean data. This can significantly reduce the amount of memory required to store

Boolean variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.

- **Compact nested if-else using logical AND/OR operators** — Improves readability of generated code by compacting nested if-else statements using logical AND (`&&`) and OR (`||`) operators.

  For example, the generated code

```
if(c1) {
    if(c1) {
        a1();
    }
}
```

  now becomes

```
if(c1 && c2) {
    a1();
}
```

  and the generated code

```
if(c1) {
    /* fall through to do a1() */
}else if(c2) {
    /* fall through to do a1() */
}else{
    /* skip doing a1() */
    goto label1;
}
a1();
label1:
    a2();
```

  now becomes

```
if(c1 || c2) {
    a1();
}
a2();
```

- **Recognize if-elseif-else in nested if-else statements** — Improves readability of generated code by recognizing and emitting an `if-elseif-else` construct in place of deeply nested `if-else` statements.

  For example, the generated code

```
if(c1) {
    a1();
}else{
    if(c2) {
        a2();
    }else{
        if(c3) {
            a3();
        }
    }
}
```

  now becomes

```
if(c1) {
    a1();
}else if(c2) {
    a2();
}else if(c3) {
    a3();
}
```

- **Replace constant expressions by a single constant** — Improves readability by preevaluating constant expressions and emitting a single

constant. Furthermore, this optimization opens up opportunities for eliminating dead code.

For example, the generated code

```
if(2+3<2) {
    a1();
}else{
    a2(4+5);
}
```

now becomes

```
if(0) {
    a1();
}else{
    a2(9);
}
```

in the first phase of this optimization. The second phase eliminates the unnecessary `if` statement, resulting in

```
a2(9);
```

- **Minimize array reads using temporary variables** — In certain microprocessors, global array read operations are more expensive than accessing a temporary variable on stack. Using this option minimizes array reads by using temporary variables when possible.

  For example, the generated code

  ```
  a[i] = foo();
  if(a[i]<10 && a[i]>1) {
      y = a[i]+5;
  }else{
      z = a[i];
  }
  ```

  now becomes

  ```
  a[i] = foo();
  temp = a[i];
  if(temp<10 && temp>1) {
      y = temp+5;
  }else{
      z = temp;
  }
  ```

- **Preserve symbol names** — (See note below before using.) Preserve symbol names (names of states and data when generating code. This option is useful when the target contains custom code that accesses Stateflow data.

  This option can generate duplicate C symbols if the source chart contains duplicate symbols, for example, two substates with identical names. Enable the next option to avoid duplicate substate names.
- **Append symbol names with parent names** — (See note below before using.) Generate a state or data name by appending the name of the item's parent to the item's name.
- **Use chart names with no mangling** — (See note below before using.) Preserve the names of chart entry functions so that they can be invoked by user-written C code.

Note When you use the option **Preserve symbol names**, **Append symbol names with parent names**, or **Use chart names with no mangling**, the names of symbols in generated code are not mangled to make them unique. Because these options do not check for symbol conflicts in generated code, use them only when your symbol names are unique within the model. Conflicts in generated names cause errors such as variable aliasing and compilation errors.

---

3 Click **Apply** to apply the selected options or **OK** to apply the options and close the dialog.

Note Use the **Chart Properties** dialog to tell the simulation target builder to recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode them as C bitwise operations.

Custom Target Coder Options

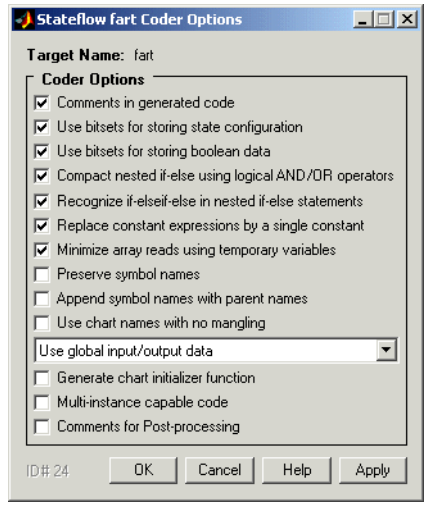
Custom targets have any other name than `sfun` or `rtw`. To specify code generation options for a custom target, do the following:

1 Open the **Stateflow Custom Target Builder** dialog.

See "Accessing the Target Builder Dialog for a Target" on page 11-9 for instructions on how to access this dialog.

2 Select the **Coder Options** button.

The **Coder Options** dialog box for the custom target appears as follows:



3 Select any or all of the following options:

For any of the following options, see "RTW Target (rtw) Coder Options" on page 11-13 for a description.

- **Comments in generated code**
- **Use bitsets for storing state configuration**
- **Use bitsets for storing boolean data**
- **Compact nested if-else using logical AND/OR operators**
- **Recognize if-elseif-else in nested if-else statements**
- **Replace constant expressions by a single constant**
- **Minimize array reads using temporary variables**

- **Preserve symbol names**
- **Append symbol names with parent names**
- **Use chart names with no mangling**

The following options are unique to custom targets:

- **I/O Format Options** — Can be any one of the following:

  Select **Use global input/output data** to generate chart input and output data as global variables.

  Select **Pack input/output data into structures** to generate structures for chart input data and chart output data.

  The **Separate argument for input/output data** generates input and output data as separate arguments to a function.
- **Generate chart initializer function** — Generates a function initializer of data.
- **Multiinstance capable code** — Generates multiply instantiable chart objects instead of a static definition.
- **Comments for Post-processing** — Internal MathWorks use only and may be discontinued in the future.

Integrating Custom Code with Stateflow Diagrams

Stateflow allows you to incorporate custom C code into Stateflow diagrams to take advantage of legacy code that augments the simulation capabilities of Simulink and Stateflow. It also allows you to define and include custom global variables that can be shared by both Stateflow generated code and your custom code.

This chapter includes sections for specifying custom code options in Stateflow and how to invoke Stateflow graphical functions from your custom code. For detailed examples, see the following:

- "Specifying Custom Code Options" on page 11-20 — Describes the target options you need to set and specify in order to build custom code into your target.
- "Calling Graphical Functions from Custom Code" on page 11-23 — Tells you how you can export graphical functions from your Stateflow charts so that they can be called from custom code that is added to your target.

Specifying Custom Code Options

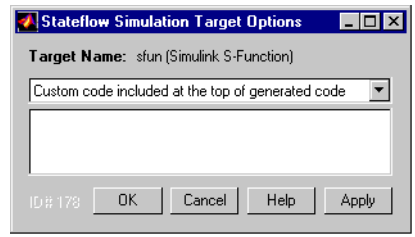
You specify configuration options to build custom code into a target with the following procedure:

1 Open the **Target Builder** dialog box.

  See "Accessing the Target Builder Dialog for a Target" on page 11-9 for instructions on how to access this dialog.

2 Select the **Target Options** button.

  The **Target Options** dialog appears similar to the following:



The Target Options dialog box identifies the target in its window title and contains a drop-down list of options for specifying the code to include in the target and where the code is located. Use the edit box following the list to add, change, and display custom code that you enter for the current option.

3 Select any or all of the following options required to specify your code and specify them in the edit box:

- **Custom code included at the top of generated code** — Custom C code to be included at the top of a generated header file that is included at the top of all generated source code files. In other words, all generated code sees code specified by this option. Use this option to include header files that declare custom functions and data used by generated code.

  Since the code specified in the **Custom code included at the top of generated code** option is included in multiple source files that are linked into a single binary, there are some limitations on what you can and cannot include. For example, you should not include global variable definitions or function bodies, such as `int x` or `void myfun(void) {...}`, through this property. This causes linking errors because these symbols are defined multiple times in the source files of the generated code. You can, however, include `extern` declarations of variables or functions such as `extern int x` or `extern void myfun(void)`.
- **Custom include directory paths** — Space-separated list of paths of directories containing custom header files to be included either directly (see previous option) or indirectly in the compiled target. See “Specifying Path Names in Custom Code Options” on page 11-22 for instructions on entering the included directory path names.
- **Custom source files** — List of source files to be compiled and linked into the target. You can separate source files with either a comma, a space, or

a new line. See "Specifying Path Names in Custom Code Options" on page 11-22 for instructions on entering the included directory path names.

- **Custom libraries** — Space-separated list of libraries containing custom object code to be linked into the target. See "Specifying Path Names in Custom Code Options" on page 11-22 for instructions on entering the included directory path names.
- **Custom initialization code** — Code statements that are executed once at the start of simulation. You can use this initialization code to invoke functions that allocate memory or perform other initializations of your custom code.
- **Custom termination code** — Code statements that are executed at the end of simulation. You can use this code to invoke functions that free memory allocated by custom code or perform other cleanup tasks.
- **Code Generation Directory** — For custom targets you can specify an optional directory to receive the generated code.

4 Click **Apply** to apply the specification to the target or **OK** to apply the specifications and close the dialog.

If you make a change in one of your custom code options, to force the rebuild of the S-function to incorporate your changes you must either change one of the charts slightly (this forces a rebuild when you simulate again) or go to the **Simulation Target Builder** dialog box and select the **Rebuild All** option.

References

For additional information on specifying custom code for your target, see the following online articles:

- "Integrating Custom C Code Using Stateflow 2.0" by V. Raghavan
- "Automatic Code Generation from Stateflow for Palm OS Handhelds: a Tutorial" by D. Maclay

Specifying Path Names in Custom Code Options

The following information applies to all custom code options that require the entry of a file or directory path.

Entries for the following custom code options can include relative or absolute paths:

- Custom include directory paths
- Custom source files
- Custom libraries
- Custom makefiles

It is not recommended that you use absolute paths for these properties, because you have to manually change them to point to new locations if you move your files and the model. Using relative path names shields you from the complexity of managing the path names as they change.

If the entered files are specified with a relative path, Stateflow searches the following directories for them:

- The current directory
- The model directory (if different from the current directory)
- The list of include directories specified for the **Custom include directory paths** (see previous option)
- All the directories on MATLAB's search path, excluding the toolbox directories

You can use the forward slash (/) or backward slash (\) as a file separator regardless of whether you are on a UNIX or PC platform. The makefile generator parses these strings and returns the path names with the correct platform-specific file separators.

Paths can also contain $...$ enclosed tokens that are evaluated in the MATLAB workspace. For example, the entry `$mydir1$\dir1`, where `mydir1` is a string variable defined in the MATLAB workspace as `'d:\work\source\module1'`, declares the directory `d:\work\source\module1\dir1` as a custom include path.

Calling Graphical Functions from Custom Code

To call a graphical function from your custom code:

1. Create the graphical function at the root level of the chart that defines the function (see "Creating a Graphical Function" on page 5-48).

2 Export the function from the chart that defines the function (see "Exporting Graphical Functions" on page 5-53).

   This option implicitly forces the chart and function names to be preserved.

3 Include the generated header file `chart_name.h` at the top of your custom code, where `chart_name` is the name of the chart that contains the graphical function.

   The chart header file contains the prototypes for the graphical functions that the chart defines.

Starting the Build

This section describes how to start a build for a simulation target (`sfun`) or Real-Time Workshop target (`rtw`) in the following topics:

- "Starting a Simulation Target Build" on page 11-25 — Tells you how to start a build for the simulation target (`sfun`).
- "Starting an RTW Target Build" on page 11-25 — Tells you how to start a build for the RTW target (`rtw`).

While building a target, Stateflow displays a stream of progress messages in the MATLAB Command Window. You can determine the success or failure of the build by examining these messages (see "Parsing Stateflow Diagrams" on page 11-27).

Starting a Simulation Target Build

You can start a target build for a simulation target (`sfun`) in one of the following ways:

- By selecting **Start** from the Stateflow or Simulink editor's **Simulation** menu

  Automatically builds and runs a simulation target.
- By selecting **Debug** from the Stateflow editor's **Tools** menu

  Automatically builds and runs a simulation target. This is equivalent to the previous method.
- By selecting the **Build** button on the **Simulation Target Builder** dialog box for the target

  Use this option if you want to build a simulation target without running it. You would typically want to do this to ensure that Stateflow can build a target containing custom code.

  Using the target builder to launch the build allows you to choose between an incremental build, a full build, and a build without code generation. See "Specifying Build Options for a Target" on page 11-10 for more information.

Starting an RTW Target Build

You can start a target build for an Real-Time Workshop (`rtw`) target in one of the following ways:

- By selecting the **Build RTW** button on the **RTW Target Builder** dialog box for the target

  You must use this option to build stand-alone targets. Using the target builder to launch the build allows you to choose between full build or rebuild and a build of Stateflow code only. See “Specifying Build Options for a Target” on page 11-10 for more information.
- By selecting the **Build** button on the RTW panel of Simulink’s **Simulation Parameters** dialog box

  Select **Real-Time Workshop** options on the **RTW Target Builder** dialog box to access the **Simulation Parameters** dialog box of Simulink. See “Specifying Build Options for a Target” on page 11-10 for more information.

Parsing Stateflow Diagrams

The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. This section describes the Stateflow parser and how its messages appear with the following topics:

- "Calling the Stateflow Parser" on page 11-27 — Shows you how to call the Stateflow Parser to parse your current diagram at any time.
- "Parsing Diagram Example" on page 11-28 — Gives you an example of parsing an example Stateflow diagram with a parsing error.

Calling the Stateflow Parser

You call the Stateflow parser in one of the following ways:

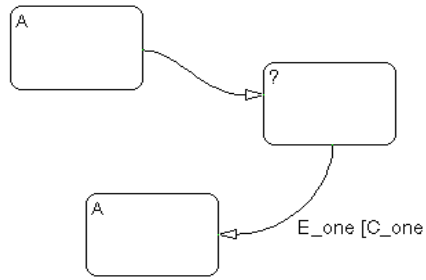
- Parse an individual Stateflow diagram in the Stateflow diagram editor by selecting **Parse Diagram** from the **Tools** menu.
- Parse a Stateflow machine, that is, all the Stateflow charts in a model, by selecting **Parse** from the **Tools** menu in the Stateflow diagram editor.
- When you simulate a model, build a target, or generate code, the Stateflow machine is automatically parsed.

In all cases, the **Stateflow Builder** window displays when parsing is complete. If parsing is unsuccessful (that is, an error is detected), the Stateflow diagram editor automatically appears with the highlighted object causing the first parse error. In the **Stateflow Builder** window, each error is displayed with a leading red button icon. You can double-click any error in this window to bring its source Stateflow diagram to the front with the source object highlighted. See "Parsing Diagram Example" on page 11-28 for example displays of parsing results in the Stateflow Builder window.

Note Parsing informational messages are also displayed in the MATLAB Command Window.

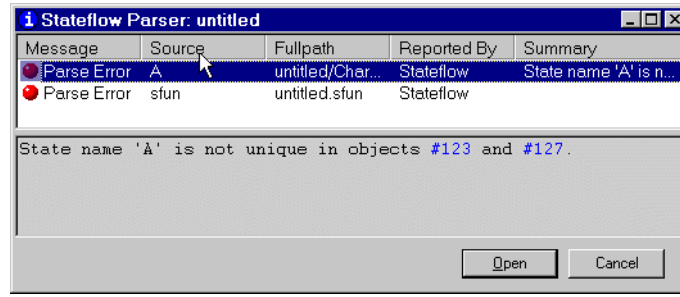
Parsing Diagram Example

These steps describe parsing, assuming this Stateflow diagram.



1 Parse the Stateflow diagram.

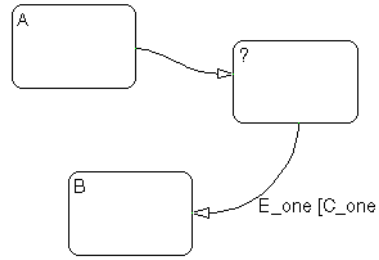
Choose **Parse Diagram** from the graphics editor **Tools** menu to parse the Stateflow diagram. State `A` in the upper left corner is selected and this message is displayed in the pop-up window and the MATLAB Command Window.



2 Fix the parse error.

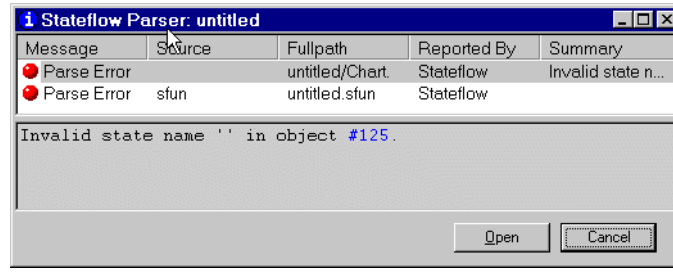
In this example, there are two states with the name `A`. Edit the Stateflow diagram and label the duplicate state with the text `B`.

The Stateflow diagram should look similar to this.



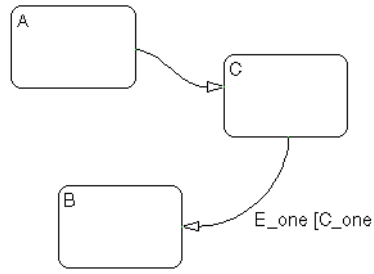
3 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message displays in the pop-up menu and the MATLAB Command Window.



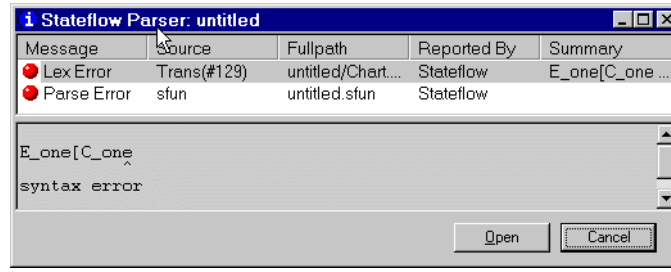
4 Fix the parse error.

In this example, the state with the question mark needs to be labeled with at least a state name. Edit the Stateflow diagram and label the state with the text `C`. The Stateflow diagram should look similar to this.



5 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up window and the MATLAB Command Window.

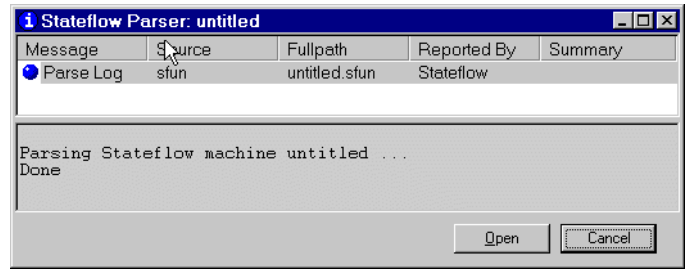


6 Fix the parse error.

In this example, the transition label contains a syntax error. The closing bracket of the condition is missing. Edit the Stateflow diagram and add the closing bracket so that the label is `E_one [C_one]`.

7 Reparse.

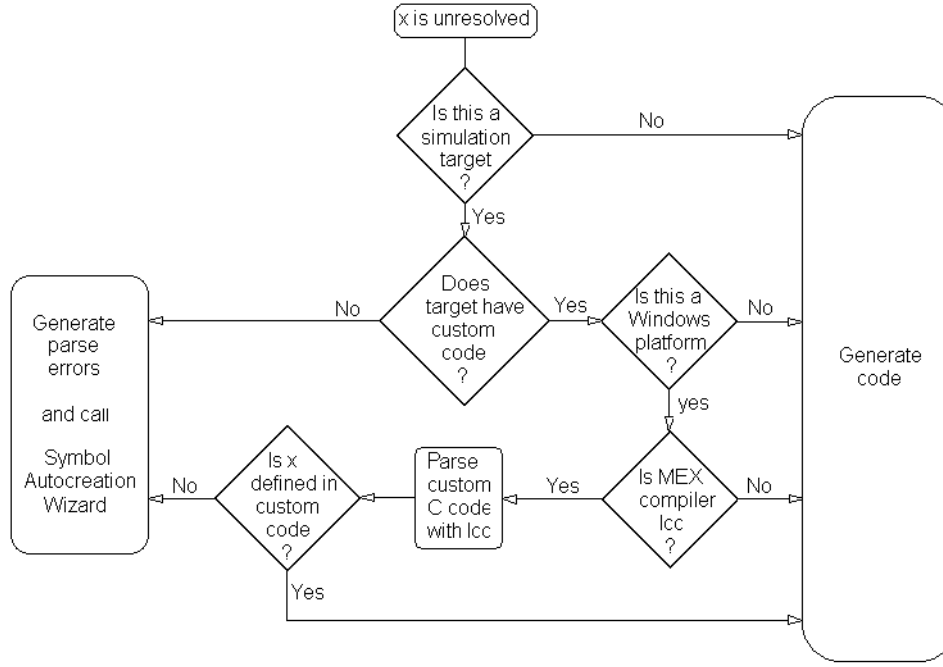
Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up window and the MATLAB Command Window.



The Stateflow diagram now has no parse errors.

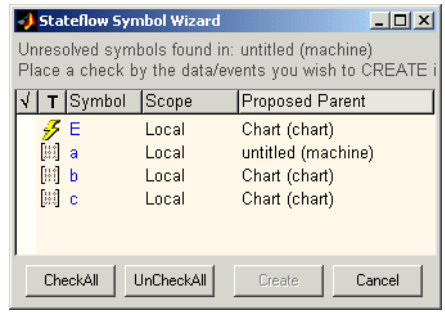
Resolving Event, Data, and Function Symbols

When you simulate a model, build a target, or generate code for a target, the Stateflow machine is automatically parsed (see "Parsing Stateflow Diagrams" on page 11-27). During that time, if Stateflow finds that your diagram does not resolve some of its symbols, it uses the following process to determine whether to report parse errors for the unresolved symbols or continue generating code:



Symbol Autocreation Wizard

The Symbol Autocreation Wizard helps you to add missing data and events to your Stateflow charts. When you parse or simulate a diagram, the Wizard detects references to data and events that are not already defined in the Stateflow Explorer and opens with a list of the recommended data or events that you need to define.



To accept, reject, or change a recommended item, do the following:

- To accept an item, select the space in front of the item under the check mark column.

  To accept all items, select the **CheckAll** button.
- To reject an item, leave it unchecked.
- To change an item, select the value under the **T** (type), **Scope**, or **Proposed Parent** column for that item.

  Each time you select the value, the Wizard replaces the entry with a different value. Keep selecting until the desired value appears.

When you are satisfied with the proposed symbol definitions, click the Wizard's **Create** button to add the symbols to Stateflow's data dictionary.

Error Messages

When building a target or parsing a diagram, you might see error messages from any of the following sources: the parser, the code generator, or external build tools (make utility, C compiler, linker). Stateflow displays errors in a dialog box and in the MATLAB Command Window. Double-clicking a message in the error dialog zooms the source Stateflow diagram to the object that caused the error.

This section contains the following topics:

- "Parser Error Messages" on page 11-34 — Lists some of the messages you can receive during parsing of your Stateflow diagram.
- "Code Generation Error Messages" on page 11-35 — Lists some of the messages you can receive during code generation for your Stateflow diagram.
- "Compilation Error Messages" on page 11-36 — Explains the difference between compilation error messages that you receive during parsing and code generation messages.

Parser Error Messages

The Stateflow parser flags syntax errors in a state chart. For example, using a backward slash (\) instead of a forward slash (/) to separate the transition action from the condition action generates a general parse error message.

Typical parse error messages include the following:

- `"Invalid state name xxx"` or `"Invalid event name yyy"` or `"Invalid data name zzz"`

  A state, data, or event name contains a nonalphanumeric character other than underscore.
- `"State name xxx is not unique in objects #yyy and #zzz"`

  Two or more states at the same hierarchy level have the same name.
- `"Invalid transition out of AND state xxx (#yy)"`

  A transition originates from an AND (parallel) state.
- `"Invalid intersection between states xxx and yyy"`

  Neighboring state borders intersect. If the intersection is not apparent, consider the state to be a cornered rectangle instead of a rounded rectangle.

- "Junction #x is sourcing more than one unconditional transition"

  More than one unconditional transition originates from a connective junction.
- "Multiple history junctions in the same state #xxx"

  A state contains more than one history junction.

Code Generation Error Messages

Typical code generation error messages include the following:

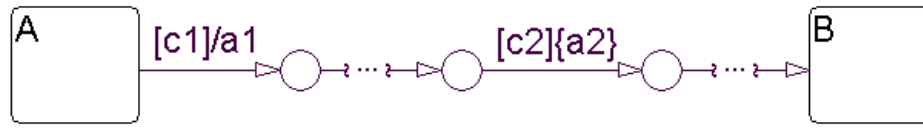
- "Failed to create file: modelName_sfun.c"

  The code generator does not have permission to generate files in the current directory.
- "Another unconditional transition of higher priority shadows transition # xx"

  More than one unconditional inner, default, or outer transition originates from the same source.
- "Default transition cannot end on a state that is not a substate of the originating state."

  A transition path starting from a default transition segment in one state completes at a destination state that is not a substate of the original state.
- "Input data xxx on left hand side of an expression in yyy"

  A Stateflow expression assigns a value to an **Input from Simulink** data object. By definition, Stateflow cannot change the value of a Simulink input.
- "Transition <number> has a condition action which is preceded by a transition <number> containing a transition action. This is not allowed as it results in out-of-order execution, i.e., the condition action of <number> gets executed before the transition action of <number>."



The preceding Stateflow diagram flags this error. Assuming that there are no other actions than those indicated for the labeled transition segments

between state A and state B, the sequence of execution that takes place when state A is active is expressed by the following pseudocode:

```
If (c1) {
   if(c2) {
       a2;
       exit A;
       a1;
       enter B;
   }
}
```

Because condition actions are evaluated when their guarding condition is true and transition actions are evaluated when the transition is actually taken, condition action `a2` is executed prior to transition action `a1`. This violates the apparent graphical sequence of executing `a1` then `a2`. In this case, the preceding diagram is flagged for an error during build time. As a remedy, the user can change `a1` and `a2` to be both condition or transition actions.

Compilation Error Messages

If compilation errors indicate the existence of undeclared identifiers, verify that variable expressions in state, condition, and transition actions are defined.

Consider, for example, an action language expression such as `a=b+c`. In addition to entering this expression in the Stateflow diagram, you must create data objects for `a`, `b`, and `c` using the Explorer. If the data objects are not defined, the parser assumes that these unknown variables are defined in the **Custom code** portion of the target (which is included at the beginning of the generated code). This is why the error messages are encountered at compile time and not at code generation time.

Generated Files

All generated files are placed in a subdirectory of the `sfprj` subdirectory of the MATLAB current directory. You set the MATLAB current directory in the **Start in** field for the properties of the MATLAB program icon that you used to start MATLAB. You can change it in MATLAB with a `cd` command as you would in DOS or UNIX.

Note Do not confuse the `sfprj` subdirectory for generated files with the `sfprj` subdirectory of the project directory (the directory where the project file resides). The latter is used for project data only.

This section contains the following topics:

- "DLL Files" on page 11-37 — Explains the origin of `.dll` files that you build as part of building a simulation target.
- "Code Files" on page 11-38 — Describes the code files that you build for your target as part of code generation.
- "Makefiles" on page 11-39 — Describes the makefiles that result when you build your target into an executable.

DLL Files

If you have a Simulink model named `mymodel.mdl`, which contains two Stateflow blocks named `chart1` and `chart2`, this means that you have a machine named `mymodel` that parents two charts named `chart1` and `chart2`.

When you simulate the Stateflow chart for `mymodel.mdl`, you actually generate code for `mymodel.mdl` that is compiled into an S-function DLL file known as a simulation target. On Windows PC platforms this file is named `mymodel_sfun.dll`. On UNIX platforms the DLL file is named `mymodel_sfun.$mexext$` where `$mexext$` is `sol2` on Solaris, `mexsg` on SGI, `mexlx` on Linux, and so on.

Code Files

Code files for the Simulation target (`sfun`) are generated for each model and placed in the subdirectory `sfprj/build/<model>/sfun/src` of the current directory, where `<model>` represents the name of the model.

Note Do not keep any of your custom source files in the `sfprj` subdirectory of your model directory.

The code generated for the simulation target `sfun` is organized into the following files:

- `<model>_sfun.h` is the machine header file. It contains the following:
  - All the defined global variables needed for the generated code
  - Type definition of the Stateflow machine-specific data structure that holds machine-parented local data
  - External declarations of any Stateflow machine-specific global variables and functions
  - Custom code strings specified via the **Target Options** dialog box
- `<model>_sfun.c` is the machine source file. It includes the machine header file and all the chart header files (described below) and contains the following:
  - All the machine-parented event broadcast functions
  - Simulink interface code
- `<model>_sfun_registry.c` is a machine registry file that contains Simulink interface code.
- `<model>_sfun_cn.h` is the chart header file for the chart `chartn`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has (see the following note). This file contains type definitions of the chart-specific data structures that hold chart-parented local data and states.
- `<model>_sfun_cn.c` is the chart source file for `chartn`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has (see the following note). This chart source file includes the machine header file and the corresponding chart header file. It contains the following:

- Chart-parented data initialization code
- Chart execution code (state entry, during, and exit actions, and so on)
- Chart-specific Simulink interface code

Note Every chart is assigned a unique number at creation time by Stateflow. This number is used as a suffix for the chart source and chart header file names for every chart (where `n` = 1, 2, 3, and so on, depending on how many charts your model has).

Makefiles

Makefiles generated for your model are platform and compiler specific. On UNIX platforms, Stateflow generates a `gmake`-compatible makefile named `mymodel_sfun.mku` that is used to compile all the generated code into an executable. On PC platforms, an ANSI C compiler-specific makefile is generated based on your C-MEX setup as follows:

- If your installed compiler is Microsoft Visual C++ 4.2, 5.0, or 6.0, the following files are generated:
  - MSVC-compatible makefile named `mymodel_sfun.mak`
  - Symbol definition file named `mymodel_sfun.def` (required for building S-function DLLs)
- If your installed compiler is Watcom 10.6 or 11.0, the following file is generated:
  - Watcom-compatible makefile named `mymodel_sfun.wmk`
- If your installed compiler is Borland 5.0, the following files are generated:
  - Borland-compatible makefile named `mymodel_sfun.bmk`
  - Symbol definition file named `mymodel_sfun.def` (required for building S-function DLLs)
- If you choose `lcc-win32`, a bundled ANSI-C compiler shipped with Stateflow, the following file is generated:
  - An `lcc` compatible makefile named `mymodel_sfun.lmk`

Stateflow Coder also generates another support file needed for the make process, named `<model>_sfun.mol`.

12

Debugging and Testing

To ensure that your Stateflow diagrams are behaving as you expect them to, use the Stateflow debugger to evaluate code coverage and perform dynamic checking during simulation.

Overview of the Stateflow Debugger (p. 12-2)	Describes the Stateflow debugger that you used to evaluate code coverage and perform dynamic checking during simulation.
Stateflow Debugger User Interface (p. 12-5)	Describes the parts of the Debugger window during debugging.
Debugging Run-Time Errors Example (p. 12-11)	Shows you how to debug run-time errors in Stateflow diagrams with an actual example model.
Debugging State Inconsistencies (p. 12-16)	Describes how state inconsistencies due to faulty Stateflow notation are detected and debugged.
Debugging Conflicting Transitions (p. 12-18)	Describes how conflicting transitions, transitions that are equally valid during execution, are detected and debugged.
Debugging Data Range Violations (p. 12-20)	Describes how to debug for occurrences of the value of a data object exceeding its maximum value or dropping below its minimum value.
Debugging Cyclic Behavior (p. 12-22)	Describes how the Debugger detects algorithms that lead to infinite recursions and looping caused by event broadcasts.
Stateflow Chart Model Coverage (p. 12-26)	Describes how the Model Coverage tool determines the extent to which a model test case exercises simulation control flow paths through a model.

Overview of the Stateflow Debugger

Use the Stateflow debugger to evaluate code coverage and perform dynamic checking during simulation to ensure that your Stateflow diagrams are behaving exactly as you expect them to.

Note Generally speaking, debugging options should be disabled for Real-Time Workshop and stand-alone targets. The Debugger does not interact with Real-Time Workshop or stand-alone targets and the overhead incurred from the added instrumented code is undesirable.

This section contains the following topics:

- "Typical Debugging Tasks" on page 12-2 — Lists some of the debugging tasks you want to accomplish with the Debugger during simulation.
- "Including Error Checking in the Target Build" on page 12-3 — Shows you the options that you can add to generated code for your target to help you debug your application.
- "Breakpoints" on page 12-3 — Lists execution points that you can specify in your target where execution stops for debugging purposes.
- "Run-Time Debugging" on page 12-4 — Lists debugging options that you can enable and disable during simulation.

Note When you save the Stateflow diagram, all the Debugger settings (including breakpoints) are saved.

Typical Debugging Tasks

These are some typical debugging tasks you might want to accomplish:

- Animate Stateflow diagrams, set breakpoints, and debug run-time errors
- Evaluate coverage
- Check for errors involving the following:
  - State inconsistencies

- Conflicting transitions
- Data range violations
- Cyclic behavior

Including Error Checking in the Target Build

The following error-checking options require the addition of supporting code to the generated target code through dialog selections:

- State inconsistency — See "Debugging State Inconsistencies" on page 12-16.
- Transition conflict — See "Debugging Conflicting Transitions" on page 12-18.
- Data range violations — See "Debugging Data Range Violations" on page 12-20.
- Cyclic behavior — See "Debugging Cyclic Behavior" on page 12-22.

To include the supporting code for these debugging options, select the **Enable debugging/animation** check box in the **Coder Options** dialog (see "Specifying Code Generation Options" on page 11-11). You access the **Coder Options** dialog through the **Target Builder** properties dialog box. See "Configuring a Target" on page 11-7 for more information.

Note You must rebuild the target for any changes to any of the settings referred to above to take effect.

Breakpoints

A breakpoint indicates a point at which the Debugger halts execution of a simulating Stateflow diagram. The Debugger supports global and local breakpoints. Global breakpoints halt execution on any occurrence of the specific type of breakpoint. Local breakpoints halt execution on a specific object instance. When simulation execution halts at a breakpoint, you can

- Examine the current status of the Stateflow diagram
- Step through the execution of the Stateflow diagram
- Specify display of one of the following:

  - Call stack
  - Code coverage
  - Data values
  - Active states

The breakpoints can be changed during run-time and are immediately enforced. When you save the Stateflow diagram, all the debugger settings (including breakpoints) are saved, so that the next time you open the model, the breakpoints are as you left them.

Run-Time Debugging

Once the target is built with the debugging code, you can enable or disable the associated run-time options in the Debugger window, which affects the Debugger output display results. Enabling/disabling the options in the Debugger window affects the target code and can cause the target to be rebuilt when you start the simulation from the debugger.

There are also some run-time debugging options that do not require supporting code in the target. These options can be dynamically set:

- Enable/disable cycle detection in the Debugger window.
- Set any of the following global breakpoints through the **Breakpoints** section of the main Debugger window (see "Breakpoint Controls" on page 12-7):
  - Any chart entry
  - Any event broadcast
  - Any state entry
- Enable/disable local Debugger breakpoints at specific chart or state action execution points in these property dialog boxes:
  - Chart (see "Specifying Chart Properties" on page 5-78)
  - State (see "Setting Event Properties" on page 6-4)
- Enable/disable local Debugger breakpoints at a specific transition (either when the transition is tested or when it is determined to be valid) in the **Transition property** dialog box (see "Changing Transition Properties" on page 5-44).
- Enable/disable local Debugger breakpoints based on a specific event broadcast (see "Setting Event Properties" on page 6-4).

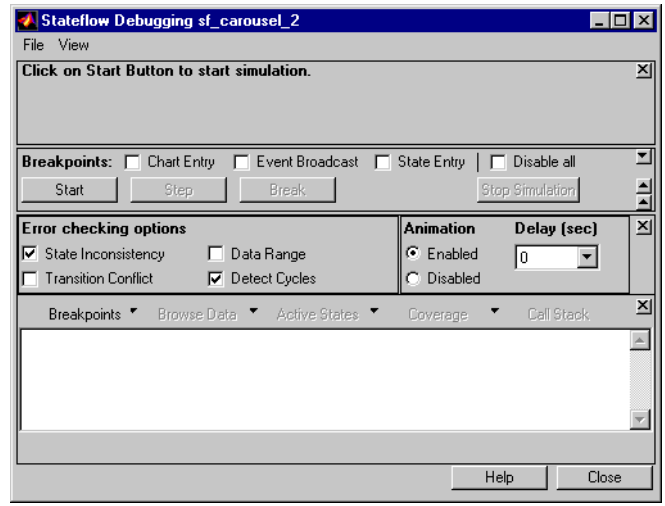
Stateflow Debugger User Interface

This section contains the following topics:

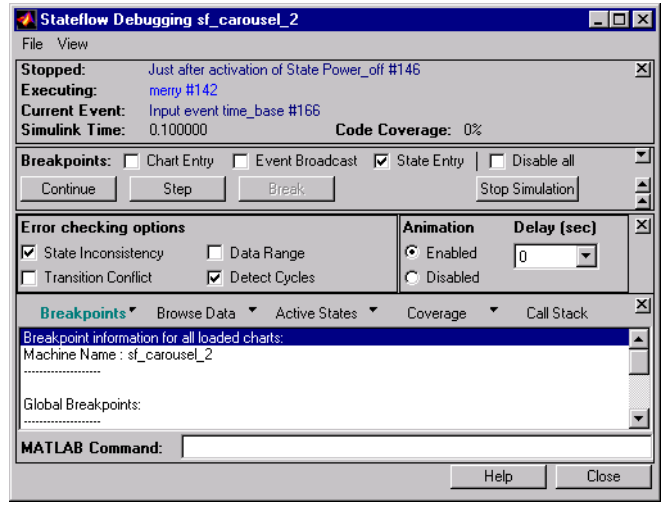
- "Debugger Main Window" on page 12-5 — Shows you the parts of the Debugger window.
- "Status Display Area" on page 12-7 — Lists and describes the items displayed in the Status Display Area of the Debugger window when a breakpoint is encountered during simulation.
- "Breakpoint Controls" on page 12-7 — Lists and describes breakpoints that you can set while simulating your application.
- "Debugger Action Control Buttons" on page 12-7 — Describes the control buttons that you can use to control execution during simulation before and after breakpoints are encountered.
- "Animation Controls" on page 12-9 — Tells you how you can enable or disable animation of your Stateflow diagram during simulation.
- "Display Controls" on page 12-9 — Describes the buttons you use during simulation to display information you use to debug your application.

Debugger Main Window

Before you begin debugging, select **Debug** from the **Tools** menu to receive the Debugger main window, which appears as shown:


Stateflow Debugging sf_carousel_2
File View
Click on Start Button to start simulation.
Breakpoints: Chart Entry Event Broadcast State Entry Disable all
Start Step Break Stop Simulation
Error checking options
State Inconsistency Data Range
Transition Conflict Detect Cycles
Animation Delay (sec)
Enabled 0
Disabled
Breakpoints Browse Data Active States Coverage Call Stack
Help Close

After you select the **Start** button to start a debugging simulation session and a breakpoint that you set is encountered, the Debugger main window appears like the following:


Stateflow Debugging sf_carousel_2
File View
Stopped: Just after activation of State Power_off #146
Executing: merry #142
Current Event: Input event time_base #166
Simulink Time: 0.100000 Code Coverage: 0%
Breakpoints: Chart Entry Event Broadcast State Entry Disable all
Continue Step Break Stop Simulation
Error checking options
State Inconsistency Data Range
Transition Conflict Detect Cycles
Animation Delay (sec)
Enabled 0
Disabled
Breakpoints Browse Data Active States Coverage Call Stack
Breakpoint information for all loaded charts:
Machine Name : sf_carousel_2
Global Breakpoints:
MATLAB Command:
Help Close

Status Display Area

Once a debugging session is in progress and a breakpoint is encountered, the following status items are displayed in the upper portion of the Debugger window:

- The currently executing model is displayed in the **Executing** field.
- The execution point that the Debugger is halted at is displayed in the **Stopped** field.

  Consecutive displays of this field show each semantic step being executed.
- The event being processed is displayed in the **Current Event** field.
- The current simulation time is displayed in the **Simulink Time** field.
- The percentage of code that has been covered thus far in the simulation is displayed in the **Code Coverage** field.

Breakpoint Controls

Use the **Breakpoint** controls to specify global breakpoints. When a global breakpoint is encountered during simulation, execution stops and the Debugger takes control. Select any or all of the following breakpoints:

- **Chart Entry** — Simulation halts on chart entry.
- **Event Broadcast** — Simulation halts when an event is broadcast.
- **State Entry** — Simulation halts when a state is entered.

These breakpoints can be changed during run-time and are immediately enforced. When you save a Stateflow diagram, the breakpoint settings are saved with it.

Debugger Action Control Buttons

Use these buttons when debugging a Stateflow machine to control the Debugger's actions:

- **Start**— Specify that simulation execution proceed until a breakpoint (global or local) is reached. Once the **Start** button has been clicked, the Stateflow diagram is marked read-only. The appearance of the graphics editor toolbar and menus changes so that object creation is not possible. When the graphics editor is in this read-only mode, its condition is referred to as *iced*.

- **Continue**— After simulation has been started, and a breakpoint has been encountered, the **Start** button is marked **Continue**. Press **Continue** to continue simulation.
- **Step** — Single step through the simulation execution.
- **Break** — Suspend the simulation and transfer control to the debugger.
- **Stop Simulation** — Stop the simulation execution and relinquish debugging control. Once the debug session is stopped, the graphics editor toolbar and menus return to their normal appearance and operation so that object creation is again possible.

Error Checking Options

The options in the **Error checking options** section of the Debugger insert generated code in the simulation target to provide breakpoints to catch different types of errors that might occur during simulation. Select any or all of the following error checking options:

- **State inconsistency —** Check for state inconsistency errors that are most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition. See “Debugging State Inconsistencies” on page 12-16.
- **Transition Conflict —** Check whether there are two equally valid transition paths from the same source at any step in the simulation. See “Debugging Conflicting Transitions” on page 12-18.
- **Data Range —** Check whether the minimum and maximum values you specified for a data in its properties dialog are exceeded. Also check whether fixed-point data overflows its base word size. See “Debugging Data Range Violations” on page 12-20.
- **Detect Cycles —** Check whether a step or sequence of steps indefinitely repeats itself. See “Debugging Cyclic Behavior” on page 12-22.

To include the supporting code designated for these debugging options, select the **Enable debugging/animation** check box in the **Coder Options** dialog. You access this dialog through the **Target Builder** properties dialog box (see “Specifying Code Generation Options” on page 11-11).

Note You must rebuild the target for any changes to any of the settings referred to above to take effect.

Animation Controls

Activating animation causes visual color changes (objects are highlighted in the selection color) in the Stateflow diagram based on the simulation execution.

Activate animation by turning on the **Enabled** check box. Deactivate animation by turning on the **Disabled** check box. You can specify the animation speed from a range of 0 (fast, the default) to 1 (slow) second.

Display Controls

Use these buttons to control the output display:

- **Call Stack** — Display a sequential list of the **Stopped** and **Current Event** status items that occur when you single-step through the simulation.
- **Coverage** — Display the current percentage of unprocessed transitions, states, and so on, at that point in the simulation. Click the button's drop-down list icon to display a list of coverage options: coverage for the current chart only, for all loaded charts, or for all charts in the model.
- **Browse Data** — Display the current values of any defined data objects.

  Fixed-point data display both their quantized integer values (stored integer) and the scaled real-world (actual) values. See "Fixed-Point Data" on page 7-14.
- **Active States** — Display a list of active states in the display area. Double-clicking any state causes the graphics editor to display that state. The drop-down list button on the **Active States** button lets you specify the extent of the display: active states in the current chart only, in all loaded charts, or for all charts in the model.
- **Breakpoints** — Display a list of the set breakpoints. The drop-down list button on the **Breakpoints** button lets you specify the extent of the display: breakpoints in the current chart only or in all loaded charts.

Once you select an output display button, that type of output is displayed until you choose a different display type. You can clear the display by selecting **Clear Display** from the Debugger's **File** menu.

Debugging Run-Time Errors Example

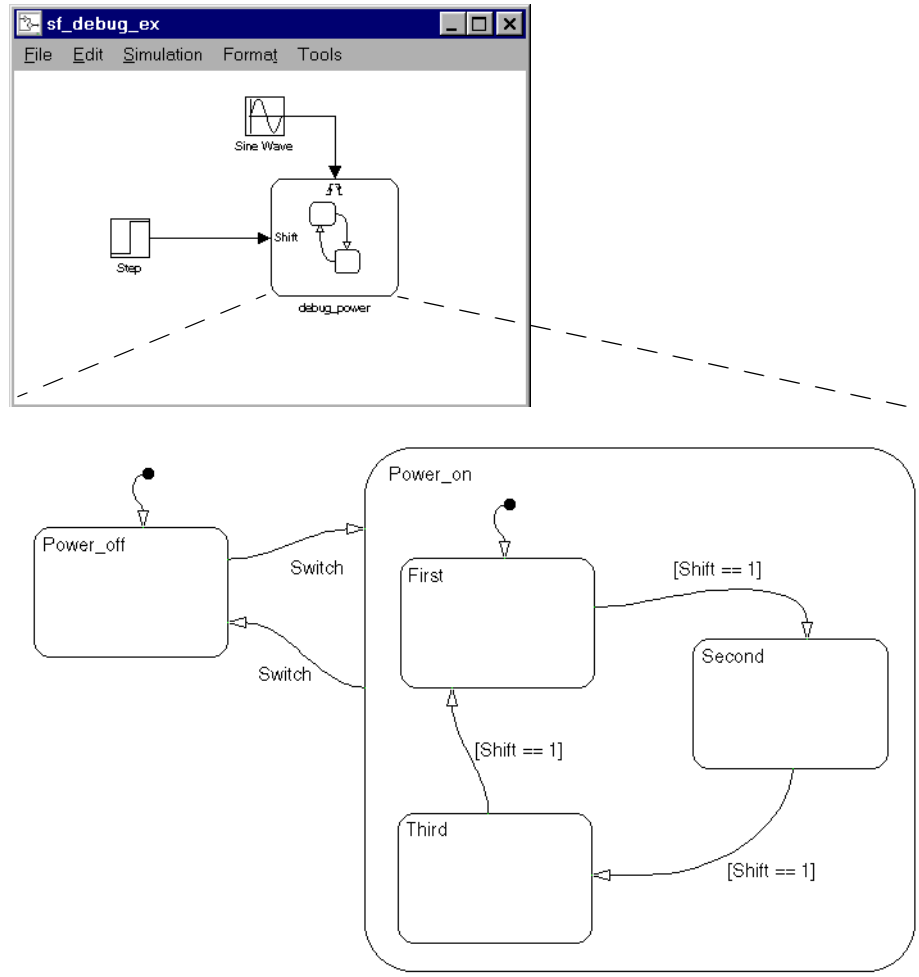
The following topics describe the steps used in a typical debugging scenario to resolve run-time errors in an example model:

1. "Create the Model and Stateflow Diagram" on page 12-11 —Create a Simulink model with a Stateflow diagram to debug.
2. "Define the sfun Target" on page 12-13 — Gives the right settings for the simulation target (`sfun`).
3. "Invoke the Debugger and Choose Debugging Options" on page 12-13 — Helps you set up the debugger for simulation.
4. "Start the Simulation" on page 12-13 — Tells you how to start simulation and what happens when you do.
5. "Debug the Simulation Execution" on page 12-14 — Shows you how to monitor the behavior of the simulating Stateflow diagram.
6. "Resolve Run-Time Error and Repeat" on page 12-14 — Shows you how to deal with the run-time error you encounter.

See "Debugged Stateflow Diagram" on page 12-14 to inspect the debugged Stateflow diagram example.

Create the Model and Stateflow Diagram

To learn how to debug some typical run-time errors, create the following Simulink model and Stateflow diagram:



Using the graphics editor **Add** menu, add the `Switch` **Input from Simulink** event and the `Shift` **Input from Simulink** data object.

The Stateflow diagram has two states at the highest level in the hierarchy, `Power_off` and `Power_on`. By default `Power_off` is active. The event `Switch` toggles the system between the `Power_off` and `Power_on` states. `Switch` is defined as an **Input from Simulink** event. `Power_on` has three substates,

First, Second, and Third. By default, when Power_on becomes active, First also becomes active. Shift is defined as an **Input from Simulink** data object. When Shift equals 1, the system transitions from First to Second, Second to Third, Third to First, and then the pattern repeats.

In the Simulink model, there is an event input and a data input. A Sine wave block is used to generate a repeating input event that corresponds with the Stateflow event Switch. The Step block is used to generate a repeating pattern of 1 and 0 that corresponds with the Stateflow data object Shift. Ideally, the Switch event occurs in a frequency that allows at least one cycle through First, Second, and Third.

Define the *sfun* Target

Choose **Open Simulation Target** from the **Tools** menu of the graphics editor. Ensure that **Enable Debugging/Animation** is enabled (checked). Click **Close** to apply the changes and close the dialog box.

Invoke the Debugger and Choose Debugging Options

Choose **Debug** from the **Tools** menu of the graphics editor. Click the **Chart entry** option under the **Break Controls** border. When the simulation begins, execution halts on entry into the chart. Click **Enabled** under the **Animation** border to turn animation on.

Start the Simulation

Click the **Start** button to start the simulation. Informational messages are displayed in the MATLAB Command Window. The graphics editor toolbar and menus change appearance to indicate a read-only interface. The Stateflow diagram is parsed, the code is generated, and the target is built. Because you specified a breakpoint on chart entry, the execution stops at that point and the Debugger display status indicates the following.

```
Stopped: Just after entering during function of Chart debug__power

Executing: sf_debug_ex_debug_power

Current Event: Input event Switch
```

Debug the Simulation Execution

At this point, you can single-step through the simulation and see whether the behavior is what you expect. Click the **Step** button and watch the Stateflow diagram animation and the Debugger status area to see the sequence of execution.

Single-stepping shows that the desired behavior is not occurring. The transitions from `Power_on.First` to `Power_on.Second`, etc., are not occurring because the transition from `Power_on` to `Power_off` takes priority. The output display of code coverage also confirms this observation.

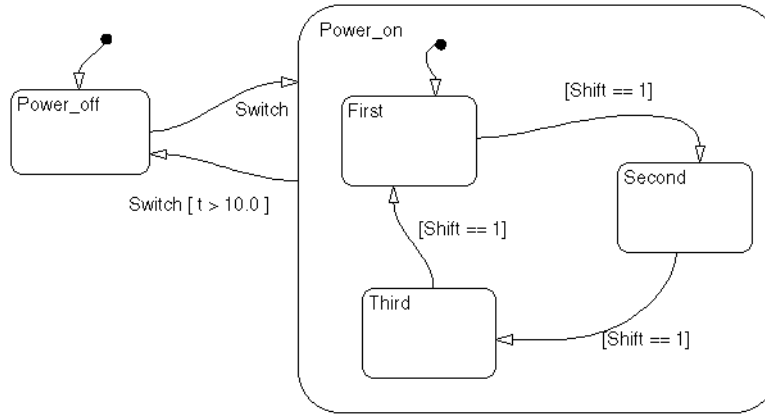
Resolve Run-Time Error and Repeat

Choose **Stop** from the **Simulation** menu of the graphics editor. The Stateflow diagram is now writeable. The generation of event `Switch` is driving the simulation and the simulation time is passing too quickly for the input data object `Shift` to have an effect. The model might need to be completely rethought.

In the meantime, there is a test that verifies the conclusion. Modify the transition from `Power_on` to `Power_off` to include a condition. The transition is not to be taken until simulation time is greater than 10.0. Make this modification and click the **Start** button to start the simulation again. Repeat the debugging single-stepping and observe the behavior.

Debugged Stateflow Diagram

This is the corrected Stateflow diagram with the condition added to the transition from `Power_on` to `Power_off`.


Power_on
Power_off
Switch
Switch [t > 10.0]
First
[Shift == 1]
Second
[Shift == 1]
Third
[Shift == 1]

Debugging State Inconsistencies

Stateflow notations specify that states are consistent if

- An active state (consisting of at least one substate) with XOR decomposition has exactly one active substate.
- All substates of an active state with AND decomposition are active.
- All substates of an inactive state with either XOR or AND decomposition are inactive.

A state inconsistency error has occurred if, after a Stateflow diagram completes an update, the diagram violates any of the preceding notation rules.

Causes of State Inconsistency

State inconsistency errors are most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition.

Design errors in complex Stateflow diagrams can also result in state inconsistency errors. These errors are only detectable using the Debugger at run-time.

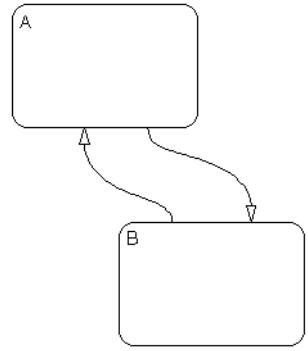
Detecting State Inconsistency

To detect the state inconsistency during a simulation:

1. Build the target with debugging enabled.
2. Invoke the Debugger and enable **State Inconsistency** checking.
3. Start the simulation.

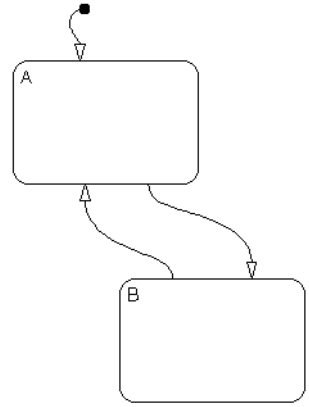
State Inconsistency Example

This Stateflow diagram has a state inconsistency.



In the absence of a default transition indicating which substate is to become active, the simulation encounters a run-time state inconsistency error.

Adding a default transition to one of the substates resolves the state inconsistency.



Debugging Conflicting Transitions

A transition conflict exists if, at any step in the simulation, there are two equally valid transition paths from the same source. In the case of a conflict, equivalent transitions (based on their labels) are evaluated based on the geometry of the outgoing transitions. See "Ordering Single Source Transitions" on page 4-8 for more information.

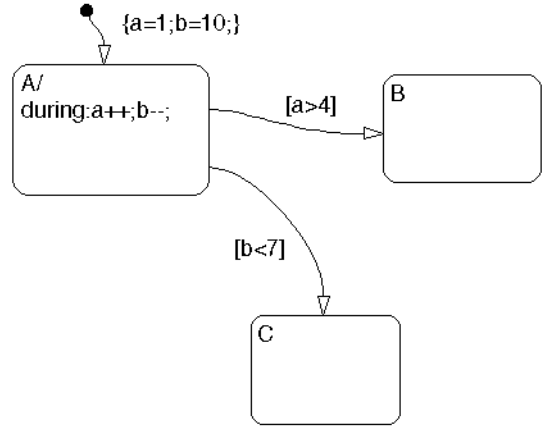
Detecting Conflicting Transitions

To detect conflicting transitions during a simulation, do the following:

1. Build the target with debugging enabled.
2. Invoke the Debugger and enable **Transition Conflict** checking.
3. Start the simulation.

Conflicting Transition Example

This Stateflow diagram has a conflicting transition.



The default transition to state `A` assigns data `a` equal to 1 and data `b` equal to 10. State `A`'s `during` action increments `a` and decrements `b`. The transition from state `A` to state `B` is valid if the condition `[a > 4]` is true. The transition from

state `A` to state `C` is valid if the condition `[b < 7]` is true. As the simulation proceeds, there is a point where state `A` is active and both conditions are true. This is a transition conflict.

Multiple outgoing transitions from states that are of equivalent label priority are evaluated in a clockwise progression starting from the twelve o'clock position on the state. In this example, the transition from state `A` to state `B` is taken.

Although the geometry is used to continue after the transition conflict, it is not recommended that you design your Stateflow diagram based on an expected execution order.

Debugging Data Range Violations

Stateflow detects the following data range violations during simulation:

- If a data object equals a value outside the range of the values set in the **Initial**, **Minimum**, and **Maximum** fields specified in the data properties dialog

  See “Setting Data Properties” on page 6-16 for a description of the **Initial**, **Minimum**, and **Maximum** fields in the data properties dialog.
- If the fixed-point result of a fixed-point operation overflows its bit size

  See “Overflow Detection for Fixed-Point Types” on page 7-37 for a description of the overflow condition in fixed-point numbers.

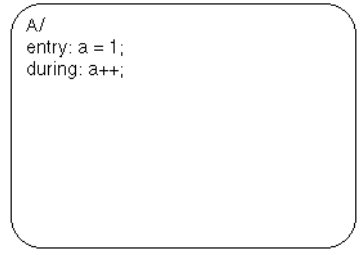
Detecting Data Range Violations

To detect data range violations during a simulation:

1. Build the target with debugging enabled.
2. Open the Debugger window.
3. In the **Error checking options** of the Debugger, select **Data Range**.
4. Start the simulation.

Data Range Violation Example

This Stateflow diagram has a data range violation.



The data a is defined to have an **Initial** and **Minimal** value of 0 and a **Maximum** value of 2. Each time an event awakens this Stateflow diagram and state A is active, a is incremented. The value of a quickly becomes a data range violation.

Debugging Cyclic Behavior

When a step or sequence of steps is indefinitely repeated (recursive), this is called cyclic behavior. The Debugger cycle detection algorithms detect a class of infinite recursions caused by event broadcasts.

To detect cyclic behavior during a simulation, do the following:

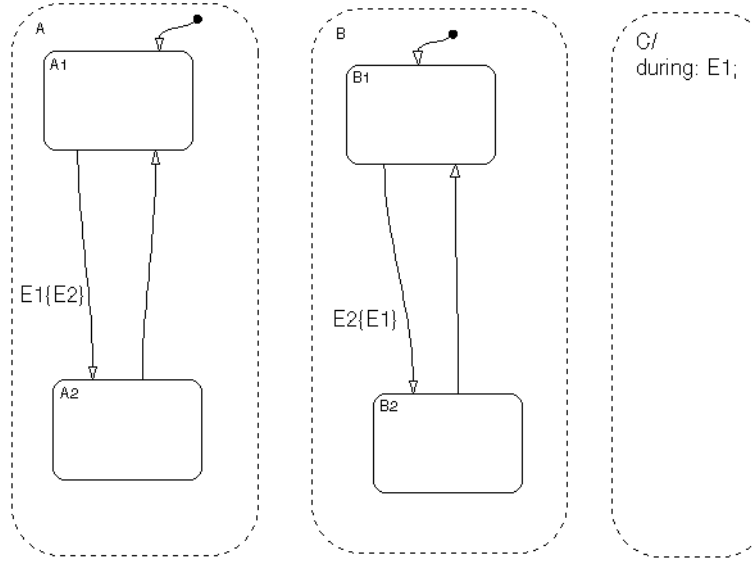
1. Build the target with debugging enabled.
2. Invoke the Debugger and enable **Detect Cycles**.
3. Start the simulation.

See the following sections for examples of cyclic behavior:

- "Cyclic Behavior Example" on page 12-23 — Shows a typical example of a cycle created by infinite recursions caused by an event broadcast.
- "Flow Cyclic Behavior Not Detected Example" on page 12-24 — Shows an example of cyclic behavior in a flow diagram that is not detected by the Debugger.
- "Noncyclic Behavior Flagged as a Cyclic Example" on page 12-25 — Shows an example of noncyclic behavior that the Debugger flags as being cyclic.

Cyclic Behavior Example

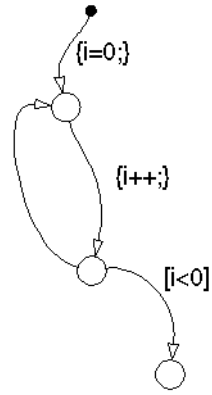
This Stateflow diagram shows a typical example of a cycle created by infinite recursions caused by an event broadcast.



When the state `C` `during` action executes, event `E1` is broadcast. The transition from state `A.A1` to state `A.A2` becomes valid when event `E1` is broadcast. Event `E2` is broadcast as a condition action of that transition. The transition from state `B.B1` to state `B.B2` becomes valid when event `E2` is broadcast. Event `E1` is broadcast as a condition action of the transition from state `B.B1` to state `B.B2`. Because these event broadcasts of `E1` and `E2` are in condition actions, a recursive event broadcast situation occurs. Neither transition can complete.

Flow Cyclic Behavior Not Detected Example

This Stateflow diagram shows an example of cyclic behavior in a flow diagram that is not detected by the Debugger.

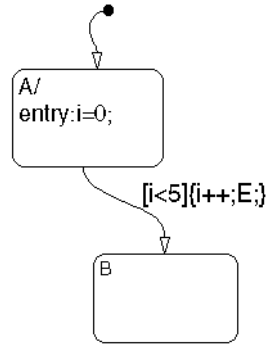


The data object `i` is set to 0 in the condition action of the default transition. `i` is incremented in the next transition segment condition action. The transition to the third connective junction is valid only when the condition `[i < 0]` is true. This condition will never be true in this flow diagram and there is a cycle.

This cycle is not detected by the Debugger because it does not involve event broadcast recursion. Detecting cycles that depend on data values is not currently supported.

Noncyclic Behavior Flagged as a Cyclic Example

This Stateflow diagram shows an example of noncyclic behavior that the Debugger flags as being cyclic.



State `A` becomes active and `i` is initialized to 0. When the transition is tested, the condition `[i < 5]` is true. The condition actions, increment `i` and broadcast event `E`, are executed. The broadcast of `E` when state `A` is active causes a repetitive testing (and incrementing of `i`) until the condition is no longer true. The Debugger flags this as a cycle when in reality the apparent cycle is broken when `i` becomes greater than 5.

Stateflow Chart Model Coverage

Model coverage is a measure of how thoroughly a model is tested. The Model Coverage tool helps you to validate your model tests by measuring model coverage for your tests. It determines the extent to which a model test case exercises simulation control flow paths through a model. The percentage of paths that a test case exercises is called its *model coverage*.

Note The Model Coverage tool is available to you if have the Simulink Performance Tools license.

For an understanding of how to generate and interpret model coverage reports for your Stateflow charts, see the following topics:

- "Making Model Coverage Reports" on page 12-26 — Gives you an overview of how model coverage reports are generated and how they are interpreted.
- "Specifying Coverage Report Settings" on page 12-27 — Gives you the settings you need to specify for each available model coverage report option.
- "Cyclomatic Complexity" on page 12-27 — Explains the cyclomatic complexity results you see on model coverage reports.
- "Decision Coverage" on page 12-28 — Explains the decision coverage results you see on model coverage reports if you select it for the report.
- "Condition Coverage" on page 12-32 — Explains the condition coverage results you see on model coverage reports if you select it for the report.
- "MCDC Coverage" on page 12-33 — Explains the MCDC (modified condition decision coverage) results you see on model coverage reports if you select it for the report.
- "Coverage Reports for Stateflow Charts" on page 12-33 — Describes different parts of a model coverage report for Stateflow charts.

Making Model Coverage Reports

Model Coverage reports are generated during simulation if you specify them (see "Specifying Coverage Report Settings" on page 12-27). For Stateflow charts, the Model Coverage tool records the execution of the chart itself and the execution of its states, transition decisions, and the individual conditions that

compose each decision. When simulation is finished, the Model Coverage report tells you how thoroughly a model has been tested, in terms of how many times each exclusive substate is entered, executed, and exited based on the history of the superstate, how many times each transition decision has been evaluated as true or false, and how many times each condition (predicate) has been evaluated as true or false.

Note You must have the Simulink Performance Tools option installed on your system to use the Model Coverage tool.

Specifying Coverage Report Settings

Coverage report settings appear in the **Coverage Settings** dialog. Access this dialog by selecting **Coverage settings** from the **Tools** menu in a Simulink model window.

Selecting the **Generate HTML Report** option on the **Coverage Settings** dialog causes Simulink to create an HTML report containing the coverage data generated during simulation of the model. Simulink displays the report in the MATLAB Help browser at the end of simulation.

Selecting the **Generate HTML Report** option also enables the selection of different coverages that you can specify for your reports. The following sections address only those coverage metrics that have direct bearing on reports for the Stateflow charts. These include decision coverage, condition coverage, and MCDC coverage. For a complete discussion of all dialog fields and entries, consult the “Model Coverage Tool” section of the Using Simulink documentation.

Cyclomatic Complexity

Cyclomatic complexity is a measure of the complexity of a software module based on its edges, nodes, and components within a control-flow graph. It provides an indication of how many times you need to test the module.

The calculation of cyclomatic complexity is as follows:

```
CC = E - N + p
```

where CC is the cyclomatic complexity, E is the number of edges, N is the number of nodes, and p is the number of components.

Within the Model Coverage tool, each decision is exactly equivalent to a single control flow node, and each decision outcome is equivalent to a control flow edge. Any additional structure in the control-flow graph is ignored since it contributes the same number of nodes as edges and therefore has no effect on the complexity calculation. This allows cyclomatic complexity to be reexpressed as follows:

```
CC = OUTCOMES - DECISIONS + p
```

For analysis purposes, each chart is considered to be a single component.

Decision Coverage

Decision coverage interprets a model execution in terms of underlying decisions where behavior or execution must take one outcome from a set of mutually exclusive outcomes.

Note Full coverage for an object of decision means that every decision has had at least one occurrence of each of its possible outcomes.

Decisions belong to an object making the decision based on its contents or properties. The following table lists the decisions recorded for model coverage

for the Stateflow objects owning them. The sections that follow the table describe these decisions and their possible outcomes.

Object	Possible Decisions
Chart	If a chart is a triggered Simulink block, it must decide whether or not to execute its block. See "Chart as a Triggered Simulink Block Decision" on page 12-29. If a chart contains exclusive (OR) substates, it must decide which of its states to execute. See "Chart Containing Exclusive OR Substates Decision" on page 12-29.
State	If a state is a superstate containing exclusive (OR) substates, it must decide which substate to execute. See "Superstate Containing Exclusive OR Substates Decision" on page 12-30. If a state has `on` *`event name`* actions (which might include temporal logic operators), the state must decide whether or not to execute the actions. See "State with On Event_Name Action Statement Decision" on page 12-32.
Transition	If a transition is a conditional transition, it must decide whether or not to exit its active source state or junction and enter another state or junction. See "Conditional Transition Decision" on page 12-32.

Chart as a Triggered Simulink Block Decision

If the chart is a triggered block in Simulink, the decision to execute the block is tested. If the block is not triggered, there is no decision to execute the block, and the measurement of decision coverage is not applicable (NA).

See "Chart as Subsystem Report Section" on page 12-36.

Chart Containing Exclusive OR Substates Decision

If the chart contains exclusive (OR) substates, the decision on which substate to execute is tested. If the chart contains only parallel AND substates, this coverage measurement is not applicable (NA).

See "Chart as Superstate Report Section" on page 12-37.

Superstate Containing Exclusive OR Substates Decision

Since a diagram is hierarchically processed from the top down, procedures such as exclusive (OR) substate entry, exit, and execution are sometimes decided by the parenting superstate.

Note Decision coverage for superstates applies to exclusive (OR) substates only. A superstate makes no decisions for its parallel (AND) substates.

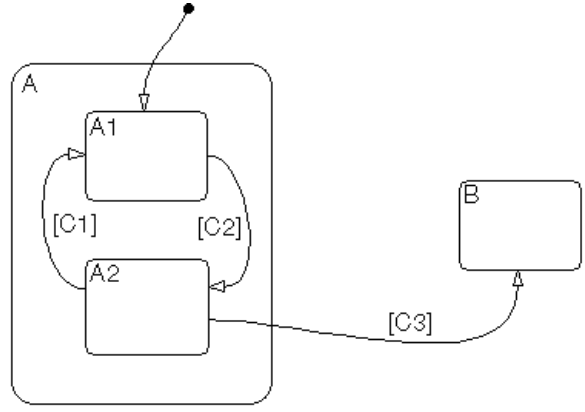
Since a superstate must decide which of its exclusive (OR) substates to process, the number of decision outcomes for the superstate is equal to the number of exclusive (OR) substates that it contains. In the examples following, the choice of which substate to process is made in one of three possible contexts.

Note Implicit transitions are shown as dashed lines in the following examples.

---

1 Active Call

In the following example, states `A` and `A1` are active.



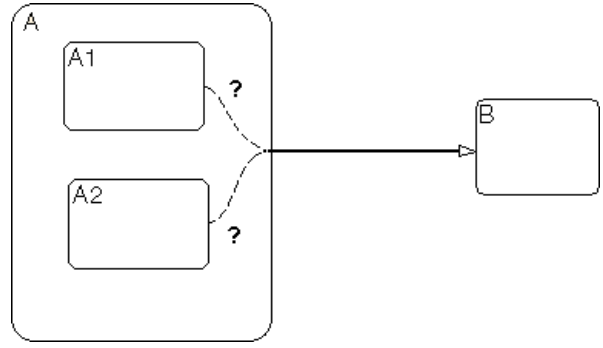
This gives rise to the following superstate/substate decisions:

- The parent of states A and B must decide which of these states to process. This decision belongs to the parent. Since A is active, it is processed.
- State A, the parent of states A1 and A2, must decide which of these states to process. This decision belongs to state A. Since A1 is active, it is processed.

  During processing of state A1, all its outgoing transitions are tested. This decision belongs to the transition and not to its parent state A. In this case, the transition marked by condition C2 is tested and a decision is made whether to take the transition to A2 or not. See “Conditional Transition Decision” on page 12-32.

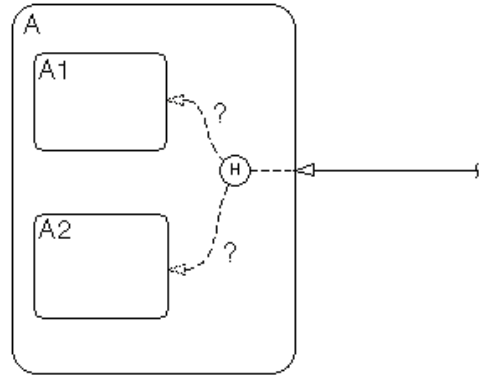
2 Implicit Substate Exit Context

In the following example, a transition takes place whose source is superstate A and whose destination is state B. If the superstate has two exclusive (OR) substates, it is the decision of superstate A as to which of these substates will perform the implicit transition from substate to superstate.



3 Substate Entry with a History Junction

A history junction, similar to the one shown in the example following, provides a superstate with the means of recording which of its substates was last active before the superstate was exited. If that superstate now becomes the destination of one or more transitions, the history junction provides it the means of deciding which previously active substate to enter.



See “State Report Section” on page 12-38.

State with On Event_Name Action Statement Decision

A state that has an `on` *`event_name`* action statement must decide whether to execute that statement based on the reception of a specified event or on an accumulation of the specified event when using temporal logic operators.

See “State Label Notation” on page 3-10 and “Using Temporal Logic” on page 7-68.

Conditional Transition Decision

A conditional transition is a transition with a triggering event and/or a guarding condition (see “Transition Label Notation” on page 3-15). In a conditional transition from one state to another, the decision to exit one state and enter another is credited to the transition itself.

See “Transition Report Section” on page 12-40.

Note Only conditional transitions receive decision coverage. Transitions without decisions are not applicable to decision coverage.

Condition Coverage

Condition coverage reports on the extent to which all possible outcomes are achieved for individual subconditions composing a transition decision.

Note Full condition coverage means that all possible outcomes occurred for each subcondition in the test of a decision.

For example, for the decision [A & B & C] on a transition, condition coverage reports on the true and false occurrences of each of the subconditions A, B, and C. This results in six possible outcomes: true and false for each of three subconditions.

See “Transition Report Section” on page 12-40.

MCDC Coverage

The modified condition decision coverage (MCDC) option reports a test’s coverage of occurrences in which changing an individual subcondition within a transition results in changing the entire transition from valid to invalid.

Note If matching true and false outcomes occur for each subcondition, coverage is 100%.

For example, if a transition executes on the condition [C1 & C2 & C3 | C4 & C5], the MCDC report for that transition shows actual occurrences for each of the five subconditions (C1, C2, C3, C4, C5) in which changing its result from true to false is able to change the result of the entire condition from true to false.

See “Transition Report Section” on page 12-40.

Coverage Reports for Stateflow Charts

The following sections of a Model Coverage report were generated by simulating the Bang-Bang Boiler demonstration model, which includes the Stateflow Chart block Bang-Bang Controller. The coverage metrics for **Decision Coverage**, **Condition Coverage**, and **MCDC Coverage** are enabled for this report; the **Look-up Table Coverage** metric is Simulink dependent and not relevant to the coverage of Stateflow charts.

This topic contains the following subtopics:

- "Summary Report Section" on page 12-34
- "Details Sections" on page 12-35
- "Chart as Subsystem Report Section" on page 12-36
- "Chart as Superstate Report Section" on page 12-37
- "State Report Section" on page 12-38
- "Transition Report Section" on page 12-40

For information on the model coverage of truth tables, see "Model Coverage for Truth Tables" on page 9-64.

Summary Report Section

Summary

Model Hierarchy/Complexity:		Test 1					
			D1		C1		MCDC
1. sf_boiler	**18**	100%		70%		40%	
2. . . . Bang-Bang Controller	**16**	100%		70%		40%	
3. . . . . SF: Bang-Bang Controller	14	100%		70%		40%	
4. . . . . SF: Heater	11	100%		70%		40%	
5. . . . . SF: On	4	100%		NA		NA	
6. . . . . SF: flash_LED	1	100%		NA		NA	
7. . . . . SF: turn_boiler	1	100%		NA		NA	
8. . . . Boiler Plant model	1	100%		NA		NA	

The Summary section shows coverage results for the entire test. It appears at the beginning of the Model Coverage report after the listing of the Start and End execution times for the test (simulation).

Each line in the hierarchy summarizes the coverage results at its level and the levels below it. It includes a hyperlink to a later section in the report with the same assigned hierarchical order number that details that coverage and the coverage of its children. See "Details Sections" on page 12-35.

The top level, sf_boiler, is the model itself. The second level, Bang-Bang Controller, is the Simulink Stateflow chart block. The next levels are superstates within the Stateflow chart control logic in order of hierarchical containment. Each of these superstates uses an SF: prefix. The bottom level, Boiler Plant model, is an additional subsystem in the model.

Details Sections

The Details heading appears directly under the “Summary Report Section” depicted in the preceding example. It precedes sections summarizing the coverage you choose to report for each object of decision in the Stateflow chart. These include the chart’s Simulink block, the chart itself, the chart’s states, and its guarded transitions. Sections on boxes and functions also appear in the Details section because those objects can contain other Stateflow objects.

An example coverage section for detailed object types appears in the following sections:

- “Chart as Subsystem Report Section” on page 12-36
- “Chart as Superstate Report Section” on page 12-37
- “State Report Section” on page 12-38
- “Transition Report Section” on page 12-40

Each of the object type sections contains the following information:

- A title containing a hyperlink to its resident diagram in Stateflow

  The chart Simulink block (Chart as Subsystem) appears in its resident Simulink model. The remaining objects appear in the chart’s Stateflow diagram.
- A hyperlink to the object’s parent
- A hyperlink to the object’s child objects
- A table summarizing decision coverage, condition coverage, and MCDC for the object with the following contents:
  - The first column lists the different coverage type for each row.
  - The second column displays coverages for the object itself.
  - The third column displays coverages for the object including sums of coverages for objects it parents (states, transitions). These coverages also appear in the Summary section.

  Results are read as a ratio of observed outcomes taken to all possible outcomes for the object. For example, if a state has four exclusive OR substates, that means that there are four possible outcomes since only one exclusive OR state can be active at a time. If only three substates are observed to execute during testing, decision coverage is reported as 75% (3/4).

- Additional tables with outcomes observed for decision coverage, condition coverage, and MCDC for the object, if applicable.

  To see these tables, see the following report sections:

  - “Chart as Superstate Report Section” on page 12-37
  - “State Report Section” on page 12-38
  - “Transition Report Section” on page 12-40

Chart as Subsystem Report Section

2. Subsystem block "Bang-Bang Controller"

Parent:	/sf_boiler
Child Systems:	Bang-Bang Controller

Metric	**Coverage (this object)**	**Coverage (inc. descendents)**
Cyclomatic Complexity	**2**	**16**
Decision (D1)	100% (2/2) decision outcomes	100% (24/24) decision outcomes
Condition (C1)	NA	70% (7/10) condition outcomes
MCDC (C1)	NA	40% (2/5) conditions reversed the outcome

The Subsystem report sees the chart as a block in a Simulink model, instead of a chart with states and transitions. You can confirm this by taking the hyperlink of the subsystem name in the title; it takes you to a highlighted Bang-Bang Controller Stateflow block sitting in its resident Simulink block diagram.

Chart as Superstate Report Section

3. Chart "Bang-Bang Controller"

Parent:	sf_boiler/Bang-Bang Controller
Child Systems:	flash_LED, turn_boiler, Heater

Metric	Coverage (this object)	Coverage (inc. descendents)
Cyclomatic Complexity	1	14
Decision (D1)	100% (2/2) decision outcomes	100% (22/22) decision outcomes
Condition (C1)	NA	70% (7/10) condition outcomes
MCDC (C1)	NA	40% (2/5) conditions reversed the outcome

Decisions analyzed:

Substate executed	100%
State "Off"	571/699
State "On"	128/699

The Chart report sees a Stateflow chart as the superstate container of all of its states and transitions. You can confirm this through the hyperlinked chart name, which takes you to a display of the control logic chart in the Stateflow diagram editor.

Cyclomatic complexity and decision coverage are also displayed for the chart and for the chart including its descendants. Condition coverage and MCDC are both not applicable (NA) coverages for a chart, but apply to descendants.

State Report Section

5. State "On"

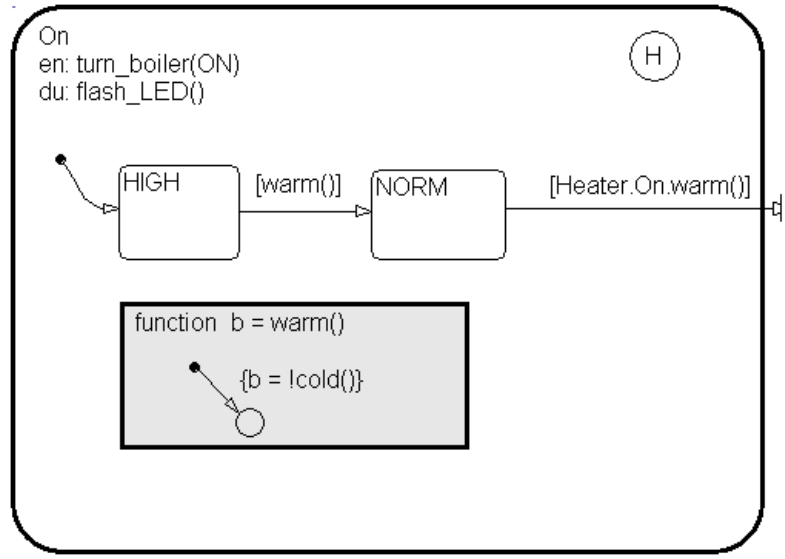
Parent: sf_boiler/Bang-Bang Controller.Heater

Metric	Coverage (this object)	Coverage (inc. descendents)
Cyclomatic Complexity	3	4
Decision (D1)	100% (6/6) decision outcomes	100% (8/8) decision outcomes

Decisions analyzed:

Substate executed	100%
State "HIGH"	88/124
State "NORM"	36/124
Substate exited when parent exits	100%
State "HIGH"	4/14
State "NORM"	10/14
Previously active substate entered due to history	100%
State "HIGH"	4/13
State "NORM"	9/13

The example state section contains a report on the state On. The Stateflow diagram for On is as follows:



On resides in the box Heater, which has its own details report (not shown) because it contains other Stateflow objects. However, because `On` is a superstate containing the two states `HIGH` and `NORM` along with a history junction and the function `warm`, it has its own numbered report in the Details section.

The decision coverage for the On state tests the decision of which of its states to execute. The results indicate that six of a possible six outcomes were tested during simulation. Each decision is described as follows:

1. The choice of which substate to execute when On is executed
2. The choice of which state to exit when On is exited
3. The choice of which substate to enter when On is entered and the History junction has a record of the previously active substate

Because each of the above decisions can result in processing either HIGH or NORM, the total possible outcomes are 3 x 2 = 6.

The decision coverage tables also display the number of occurrences for each decision and the number of times each state was chosen. For example, the first

decision was made 124 times. Of these, the HIGH state was executed 88 times and the NORM state was executed 36 times.

Cyclomatic complexity and decision coverage are also displayed for the On state including its descendants. This includes the coverage discussed above plus the decision required by the condition `[warm()]` for the transition from HIGH to NORM for a total of eight outcomes. Condition coverage and MCDC are both not applicable (NA) coverages for a state.

Note Nodes and edges that make up the cyclomatic complexity calculation have no direct relationship with model objects (states, transitions, and so on). Instead, this calculation requires a graph representation of the equivalent control flow.

Transition Report Section

Reports for transitions appear under the report sections of their owning states. They do not appear in the model hierarchy of the Summary section, since that is based entirely on superstates owning other Stateflow objects.

Transition "after(40,sec) [cold()]" from "Off" to "On"

Parent: sf_boiler/Bang-Bang Controller.Heater
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	3
Decision (D1)	100% (2/2) decision outcomes
Condition (C1)	67% (4/6) condition outcomes
MCDC (C1)	33% (1/3) conditions reversed the outcome

Decisions analyzed:

Transition trigger expression	100%
false	557/571
true	14/571

Conditions analyzed:

Description:	#1 T	#1 F
Condition 1, "sec"	571	0
Condition 2, "after(40,sec)"	14	557
Condition 3, "cold()"	14	0

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	#1 True Out	#1 False Out
Transition trigger expression		
Condition 1, "sec"	TTT	(Fxx)
Condition 2, "after(40,sec)"	TTT	TFx
Condition 3, "cold()"	TTT	(TTF)

The decision for this transition is based on the broadcast of 40 `sec` events and the condition `[cold()]`. If, after the reception of 40 `sec` events (equivalent to a 40 second delay) the environment is cold (`cold() = 1`), the decision to execute this transition and turn the Heater on is made. For other time intervals or environment conditions, the decision is made not to execute.

For decision coverage, both the true and false evaluations for the decision occurred. Because two of two decision outcomes occurred, coverage was full (that is, 100%).

Condition coverage shows that only 4 of 6 condition outcomes were tested. The temporal condition `after(40,sec)` is a short form expression for `sec[after(40,sec]` which is actually two conditions: the event `sec` and the accumulation condition `after(40,sec)`. Consequently, there are actually three conditions on the transition: `sec`, `after(40,sec)`, and `cold()`. Since each of these decisions can be true or false, there are now six possible outcomes.

A look at the **Decisions analyzed** table shows each of these conditions as a row with the recorded number of occurrences for each outcome for that decision (true or false). Decision rows in which a possible outcome did not occur are shaded. For example, the first and the third decision rows did not record an occurrence of a false outcome and are therefore shaded.

In the MC/DC report, all sets of occurrences of the transition conditions are scanned for a particular pair of decisions for each condition in which the following are true:

- The condition varies from true to false.
- All other conditions contributing to the decision outcome remain constant.
- The outcome of the decision varies from true to false, or the reverse.

For three conditions related by an implied AND operator, these criteria can be satisfied by the occurrence of the following conditions.

Condition Tested	True Outcome	False Outcome
1	**T**TT	**F**xx
2	T**T**T	T**F**x
3	TT**T**	TT**F**

Notice that in each line, the condition tested changes from true to false while the other condition remains constant. Irrelevant contributors are coded with an “x” (discussed below). If both outcomes occur during testing, coverage is complete (100%) for the condition tested.

The preceding report example shows coverage only for condition 2. The false outcomes required for conditions 1 and 3 did not occur, and are indicated by parentheses for both conditions. Therefore the table lines for conditions (rows)

1 and 3 are shaded in red. Thus, while condition 2 has been tested, conditions 1 and 3 have not and MCDC is 33%.

For some decisions, the values of some conditions are irrelevant under certain circumstances. For example, in the decision [C1 & C2 & C3 | C4 & C5] the left side of the "|" is false if any one of the conditions C1, C2, or C3 is false. The same applies to the right side result if either C4 or C5 is false. When searching for matching pairs that change the outcome of the decision by changing one condition, holding some of the remaining conditions constant is irrelevant. In these cases, the MC/DC report marks these conditions with an "x" to indicate their irrelevance as a contributor to the result. This is shown in the following example.

Transition "[c1&c2&c3 | c4&c5]" . . .

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	#1 True Out	#1 False Out
Transition trigger expression		
Condition 1, "c1"	TTTxx	FxxFx
Condition 2, "c2"	TTTxx	TFxFx
Condition 3, "c3"	TTTxx	TTFFx
Condition 4, "c4"	FxxTT	FxxFx
Condition 5, "c5"	FxxTT	FxxTF

Consider the very first matched pair. Since condition 1 is true in the **True** outcome column, it must be false in the matching **False** outcome column. This makes the conditions C2 and C3 irrelevant for the false outcome since C1 & C2 & C3 is always false if C1 is false. Also, since the false outcome is required to evaluate to false, the evaluation of C4 & C5 must also be false. In this case, a match was found with C4 = F, making condition C5 irrelevant.

13

Stateflow API

The procedures and conceptual information in this chapter explain the basic operations of the Stateflow API (Application Programming Interface). It includes the following sections:

Overview of the Stateflow API (p. 13-3)	Introduces you to concepts you need to know to understand the Stateflow API and how to use it to create and edit Stateflow diagrams.
Quick Start for the Stateflow API (p. 13-9)	Step-by-step instructions for constructing a Stateflow diagram with the Stateflow API.
Accessing the Properties and Methods of Objects (p. 13-17)	Describes the conventions used in naming the properties and methods of Stateflow API objects and the rules for using them in commands.
Displaying Properties and Methods (p. 13-19)	Information on calling built-in methods for listing properties and methods for each object type.
Creating and Destroying API Objects (p. 13-22)	Information on creating and destroying any Stateflow object with the Stateflow API, and how to connect one object with another.
Accessing Existing Stateflow Objects (p. 13-26)	Create handles to any object in an existing Stateflow diagram and use them to manipulate actual Stateflow objects in a Stateflow diagram.
Copying Objects (p. 13-29)	Learn the copy and paste procedure for copying Stateflow objects from one environment to another.
Using the Editor Object (p. 13-33)	Access the Editor object for a Stateflow diagram to perform operations that are graphical only, such as changing fonts and colors.
Entering Multiline Labels (p. 13-34)	The Stateflow API provides two techniques to enter text with more than one line for the labels of states and transitions.

Creating Default Transitions (p. 13-35)	The Stateflow API provides two means for making default transitions.
Creating a MATLAB Script of API Commands (p. 13-38)	You can execute your API commands in a single MATLAB script.

Overview of the Stateflow API

The Stateflow API is a textual programming interface with the Stateflow diagram editor from the MATLAB Command Window. Before you get started with the “Quick Start for the Stateflow API” on page 13-9, read the topics in this section for an introduction to some new concepts that are part of the Stateflow API.

- “What Is the Stateflow API?” on page 13-3 — Defines and describes the nature of the Stateflow API.
- “Stateflow API Object Hierarchy” on page 13-4 — Introduces you to the hierarchy of objects in the Stateflow API, which mimics the hierarchy of objects in Stateflow.
- “Getting a Handle on Stateflow API Objects” on page 13-6 — Introduces you to the concept of a handle that is used to represent a Stateflow API object in MATLAB.
- “Using API Object Properties and Methods” on page 13-7 — Introduces you to the properties and methods of each API object. Properties and methods do the work of the API to create and change Stateflow diagrams.
- “API References to Properties and Methods” on page 13-7 — Introduces you to the extensive reference information available in this guide on each individual property and method.

Caution You cannot undo any operation to the Stateflow diagram editor performed through the Stateflow API. If you do perform an editing operation through the API, the undo and redo buttons are disabled from undoing and redoing any prior operations.

What Is the Stateflow API?

The Stateflow Application Programming Interface (API) is a tool of convenience that lets you create or change Stateflow diagrams with MATLAB commands. By placing Stateflow API commands in a MATLAB script, you can automate Stateflow diagram editing processes in a single command.

There are many possible applications for the Stateflow API. Here are some:

- Create a script that performs common graphical edits that makes editing of Stateflow diagrams easier.
- Create a script that immediately creates a repetitive “base” Stateflow diagram.
- Create a script that produces a specialized report of your model.

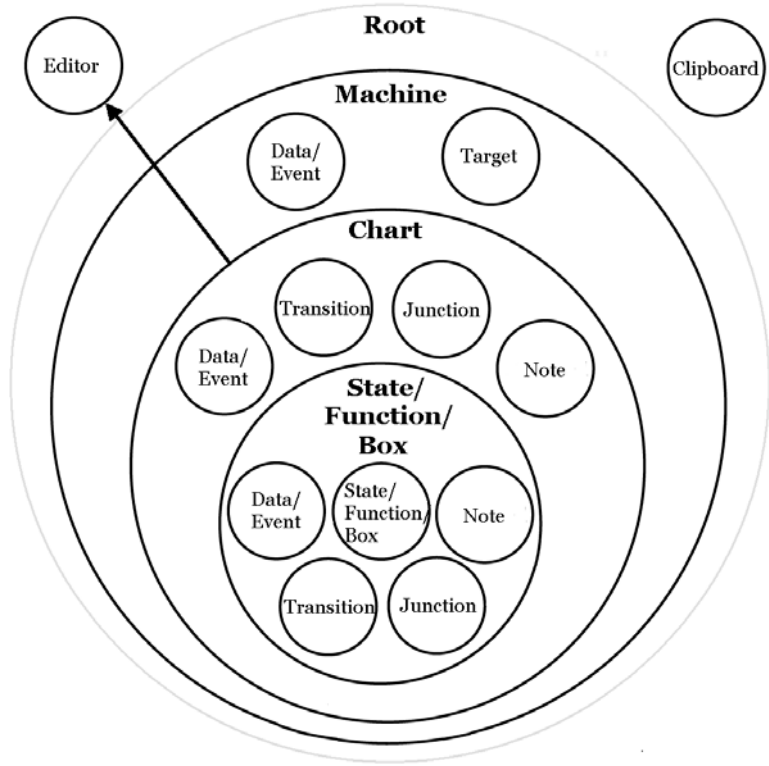
The Stateflow API consists of objects that represent actual Stateflow objects. For example, an API object of type State represents a Stateflow state, an API object of type Junction represents a Stateflow junction, and so on.

Each API object has methods and properties you use to perform editing operations on it. The correspondence between API object and Stateflow object is so close that what you do to a Stateflow API object affects the object it represents in the Stateflow diagram editor, and what you do to a graphical object in the Stateflow diagram editor affects the Stateflow API object that represents it.

The following topics introduce you to the objects, properties, and methods of the Stateflow API.

Stateflow API Object Hierarchy

Stateflow API objects represent actual Stateflow objects in a Stateflow diagram. Like Stateflow objects, API objects contain or are contained by other Stateflow objects. For example, if state A contains state B in the Stateflow diagram editor, then the API object for state A contains the API object for state B. The following diagram depicts the Stateflow API hierarchy of objects:



Rules of containment define the Stateflow and Stateflow API object hierarchy. For example, charts can contain states but states cannot contain charts. The hierarchy of Stateflow objects, also known as the Stateflow data dictionary, is depicted in the section "Overview of the Stateflow Machine" on page 10-2. The Stateflow API hierarchy is very similar to the hierarchy of the Stateflow data dictionary and consists of the following layers of containment:

- **Root** — The Root object (there is only one) serves as the parent of all Stateflow API objects. It is a placeholder at the top of the Stateflow API hierarchy to distinguish Stateflow tool objects from the objects of other tools such as Simulink and Handle Graphics. The Root object is automatically created when you load a model containing a Stateflow chart or call the method `sfnew` to create a new model with a Stateflow chart.

- **Machine** — Objects of type Machine are accessed through Stateflow's Root object. Machine objects are equivalent to Simulink models from a Stateflow perspective. They can hold objects of type Chart, Data/Event, and Target.
- **Chart** — Within any Machine object (model) there can be any number of chart objects. Within each object of type Chart, there can be objects of type State, Function, Box, Note, Data, Event, Transition, and Junction. These objects represent the components of a Stateflow chart.
- **State/Function/Box** — Nested within objects of type State, Function, and Box, there can be further objects of type State, Function, Box, Note, Junction, Transition, Data, and Event. Levels of nesting can continue indefinitely.

The preceding figure also shows two object types that exist outside the Stateflow containment hierarchy, which are as follows:

- **Editor** — Though not a part of the Stateflow containment hierarchy, an object of type Editor provides access to the purely graphical aspects of objects of type Chart. For each Chart object there is an Editor object that provides API access to the Chart object's diagram editor.
- **Clipboard** — The Clipboard object has two methods, `copy` and `pasteTo`, that use the clipboard as a convenient staging area to implement the operation of copy and paste functionality in the Stateflow API.

Getting a Handle on Stateflow API Objects

You manipulate Stateflow objects by manipulating the Stateflow API objects that represent them. You manipulate Stateflow API objects through a MATLAB variable called a *handle*.

The first handle that you require in the Stateflow API is a handle to the Root object, the parent object of all objects in the Stateflow API. In the following command, the method `sfroot` returns a handle to the Root object:

```
rt = sfroot
```

Once you have a handle to the Root object, you can find a handle to the Machine object corresponding to the Simulink model you want to work with. Once you have a handle to a machine object, you can find a handle to a Chart object for the chart you want to edit. Later on, when you create objects or find existing objects in a Stateflow chart, you receive a handle to the object that allows you to manipulate the actual object in Stateflow.

You are introduced to obtaining handles to Stateflow API objects and using them to create and alter Stateflow diagrams in "Quick Start for the Stateflow API" on page 13-9.

Using API Object Properties and Methods

Once you obtain handles to Stateflow API objects, you can manipulate the Stateflow objects that they represent through the properties and methods that each Stateflow API object possesses. You access the properties and methods of an object through a handle to the object.

API properties correspond to values that you normally set for an object through the user interface of the Stateflow diagram editor. For example, you can change the position of a transition by changing the Position property of the Transition object that represents the transition. In the Stateflow diagram editor you can click-drag the source, end, or midpoint of a transition to change its position.

API methods are similar to functions for creating, finding, changing, or deleting the objects they belong to. They provide services that are normally provided by the Stateflow diagram editor. For example, you can delete a transition in the Stateflow diagram editor by calling the `delete` method of the Transition object that represents the transition. Deleting a transition in the diagram editor is normally done by selecting a transition and pressing the **Delete** key.

Stateflow API objects have some common properties and methods. For example, all API objects have an `Id` and a `Description` property. All API objects have a `get` and a `set` method for viewing or changing the properties of an object, respectively. Most API objects also have a `delete` method. Methods held in common among all Stateflow objects are listed in the reference section "All Object Methods" on page 15-9.

Each API object also has properties and methods unique to its type. For example, a State object has a `Position` property containing the spatial coordinates for the state it represents in the chart editor. A Data object, however, has no `Position` property.

API References to Properties and Methods

When you need to know what property's value to change or what method to call to effect a change in a Stateflow chart, you can consult the following references for specific information about an individual Stateflow API property or method:

- "API Properties and Methods by Use" on page 14-1 — This reference section lists the properties and methods of the Stateflow API organized according to their type of use in Stateflow.

  For example, if you want to use the API to change the font color or style for a state, see the section "Graphical Properties" on page 14-30.
- "API Properties and Methods by Object" on page 15-1 — This reference section lists the properties and methods of the Stateflow API by their owning objects.

  For example, if you want to change a transition with a transition property in the API, see the section "Transition Properties" on page 15-44.
- "API Methods Reference" on page 16-1 — This reference section contains individual references for each method in the Stateflow API.

  These references are ordered alphabetically and provide information on the objects that they belong to, the syntax for calling them, a description of what they do, and information on their argument and return values, along with examples.

Quick Start for the Stateflow API

This section helps you create a single Stateflow chart and its member objects using the Stateflow API. It reflects the major steps in using the Stateflow API to create a Stateflow chart.

1. "Create a New Model and Chart" on page 13-9 — Teaches you by example to create a new empty Stateflow diagram in its own new machine (model).
2. "Access the Machine Object" on page 13-9 — Tells you how to access the Stateflow Machine object, which you need to access before you can access a Stateflow Chart object.
3. "Access the Chart Object" on page 13-10 — Tells you how to access the Chart object so that you can begin creating Stateflow objects in the chart you created.
4. "Create New Objects in the Chart" on page 13-11 — Gives many example commands for creating new objects in a new Stateflow chart.

Create a New Model and Chart

Create a new model by itself in MATLAB with the following steps:

1. Close down all models in Simulink.
2. Use the objectless method `sfnew` to create a new chart.

   The `sfnew` method creates a new untitled Simulink model with a new Stateflow chart in it. Do not open the Stateflow chart.

You now have only one Simulink model in memory. You are now ready to access the API Machine object that represents the model itself.

Access the Machine Object

In the Stateflow API, each model you create or load into memory is represented by an object of type Machine. Before accessing the Stateflow chart you created in the previous section, you must first connect to its Machine (model) object. However, in the Stateflow API, all Machine objects are contained by the

Stateflow API Root object, so you must use the Root object returned by the method `sfroot` to access a Machine object.

1 Use the following command to obtain a handle to the Root object:

```
rt = sfroot
```

2 Use the handle to the Root object, `rt`, to find the Machine object representing your new untitled Simulink model and assign it a handle, `m` in the following command:

```
m = rt.find('-isa','Stateflow.Machine')
```

If, instead of one model, there are several models open, this command returns an array of different machine objects that you could access through indexing (`m(1)`,`m(2)`,...). You can identify a specific machine object using the properties of each machine, particularly the `Name` property, which is the name of the model. For example, you can use the `Name` property to find a machine with the name "myModel" with the following command:

```
m = rt.find('-isa', 'Stateflow.Machine', '-and',
'Name','myModel')
```

However, since you now have only one model loaded, the object handle `m` in the command for step 2 returns the machine that you just created. You are now ready to use `m` to access the empty Stateflow chart so that you can start filling it with Stateflow objects.

Access the Chart Object

In the previous section, "Access the Machine Object", you accessed the machine object containing your new chart to return a handle to the Machine object for your new model, `m`. Perform the following steps to access the new Stateflow chart.

1 Access the new Chart object and assign it to the workspace variable `chart` as follows:

```
chart = m.findDeep('Chart')
```

In the preceding command, the `findDeep` method of the Machine object `m` returns an array of all charts belonging to that machine. Because you

created only one chart, the result of this command is the chart you created. If you created several charts, the `findDeep` method returns an array of charts that you could access through indexing (chart(1), chart(2), and so on).

You can also use standard function notation instead of dot notation for the preceding command. In this case, the first argument is the machine object handle.

```
chart = findDeep(m, 'Chart')
```

2 Open the Stateflow chart with the following API command:

```
chart.view
```

The preceding command calls the `view` method of the Chart object whose handle is `chart`. This displays the specified chart in the Stateflow diagram editor. You should now have an empty Stateflow chart in front of you. Other Stateflow API objects have `view` methods as well.

Create New Objects in the Chart

In the previous section, you created a handle to the new Chart object, `chart`. Continue by creating new objects for your chart using the following steps:

1 Create a new state in the Chart object `chart` with the following command:

```
sA = Stateflow.State(chart)
```

This command is a Stateflow API constructor for a new state in which `Stateflow.State` is the object type for a state, `chart` is a workspace variable containing a handle to the parent chart of the new state, and `sA` is a workspace variable to receive the returned handle to the new state.

An empty state now appears in the upper left-hand corner of the diagram editor.

2 Use the `chart.view` command to bring the chart diagram editor to the foreground for viewing.

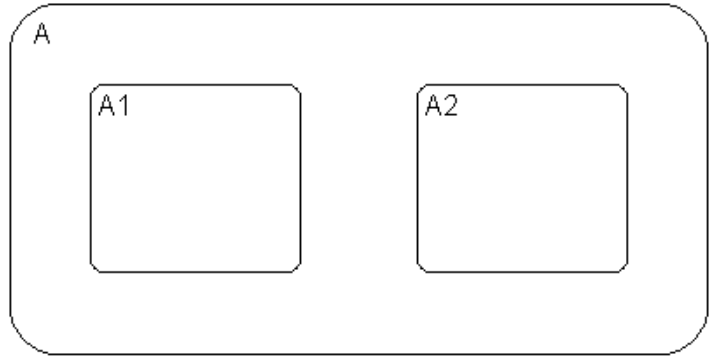
3 Assign a name and position to the new state by assigning values to the new State object's properties as follows:

```
sA.Name = 'A'
sA.Position = [50 50 310 200]
```

4 Create new states A1 and A2 inside state A and assign them properties with the following commands:

```
sA1 = Stateflow.State(chart)
sA1.Name = 'A1'
sA1.Position = [80 120 90 60]
sA2 = Stateflow.State(chart)
sA2.Name = 'A2'
sA2.Position = [240 120 90 60]
```

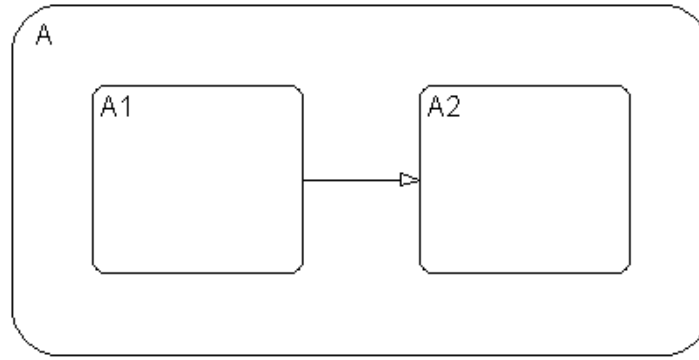
These commands create and use the workspace variables `sA`, `sA1`, and `sA2` as handles to the new states, which now have the following appearance:



5 Create a transition from the 3 o'clock position (right side) of state A1 to the 9 o'clock position (left side) of state A2 with the following commands:

```
tA1A2 = Stateflow.Transition(chart)
tA1A2.Source = sA1
tA1A2.Destination = sA2
tA1A2.SourceOClock = 3.
tA1A2.DestinationOClock = 9.
```

A transition now appears as shown:



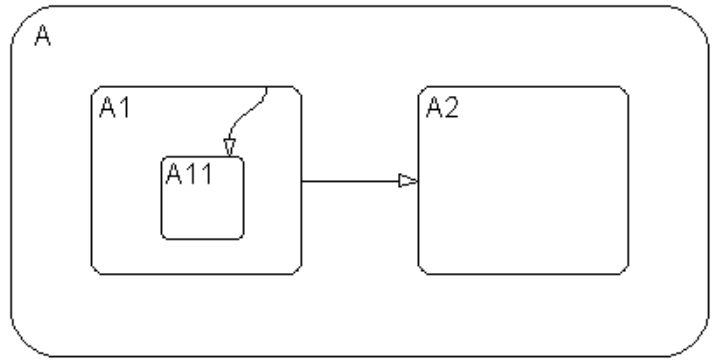
6 Draw, name, and position a new state A11 inside A1 with the following commands:

```
sA11 = Stateflow.State(chart)
sA11.Name = 'A11'
sA11.Position = [90 130 35 35]
```

7 Draw an inner transition from the 1 o'clock position of state A1 to the 1 o'clock position of state A11 with the following commands:

```
tA1A11 = Stateflow.Transition(chart)
tA1A11.Source = sA1
tA1A11.Destination = sA11
tA1A11.SourceOClock = 1.
tA1A11.DestinationOClock = 1.
```

Your Stateflow diagram now has the following appearance:



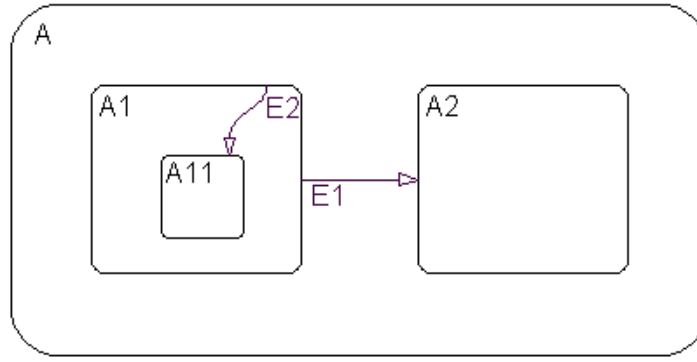
8 Add the label `E1` to the transition from state `A1` to state `A2` with the following command:

```
tA1A2.LabelString = 'E1'
```

9 Add the label `E2` to the transition from state `A1` to state `A11` with the following command:

```
tA1A11.LabelString = 'E2'
```

The Stateflow diagram now has the following appearance:



Both the state and transition labels in our example are simple one-line labels. To enter more complex multiline labels, see "Entering Multiline Labels" on page 13-34. Labels for transitions also have a `LabelPosition` property that you can use to move the labels to better locations.

10 Use the following commands to move the label for the transition from A1 to A2 to the right by 15 pixels:

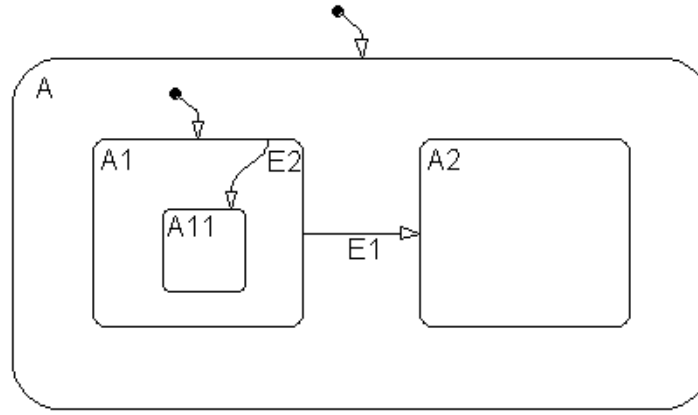
```
pos = tA1A2.LabelPosition
pos(1) = pos(1)+15
tA1A2.LabelPosition = pos
```

11 Use the following commands to finish your new chart diagram by adding default transitions to states A and A1 with source points 20 pixels above and 10 pixels to the left of the top midpoint of each state:

```
dtA = Stateflow.Transition(chart)
dtA.Destination = sA
dtA.DestinationOClock = 0
xsource = sA.Position(1)+sA.Position(3)/2-10
ysource = sA.Position(2)-20
dtA.SourceEndPoint = [xsource ysource]
dtA1 = Stateflow.Transition(chart)
dtA1.Destination = sA1
dtA1.DestinationOClock = 0
xsource = sA1.Position(1)+sA1.Position(3)/2-10
ysource = sA1.Position(2)-20
```

```
dtA1.SourceEndPoint = [xsource ysource]
```

You now have the following finished Stateflow diagram:



12 Save the Simulink model with its new Stateflow chart to the working directory as `myModel.mdl` with the following command:

```
sfsave(m.Id, 'myModel')
```

Notice that the preceding command uses the `Id` property of the Machine object `m` for saving the model under a new name.

You are now finished with the "Quick Start for the Stateflow API" section of this chapter. You can continue with the next section ("Accessing the Properties and Methods of Objects"), or you can go to the section "Creating a MATLAB Script of API Commands" on page 13-38 to see how to create a script of the API commands you used in this Quick Start section.

Accessing the Properties and Methods of Objects

All Stateflow API commands access the properties and methods of Stateflow objects. Before you start creating a new Stateflow diagram or changing an existing one, you must learn how to access the properties and methods of objects.

This section describes the conventions used in naming the properties and methods of Stateflow API objects and the rules for using them in commands.

- "Naming Conventions for Properties and Methods" on page 13-17 — Gives you a look at the conventions followed in naming properties and methods.
- "Using Dot Notation with Properties and Methods" on page 13-17 — Shows you how to use dot notation to access the value of an object's property or call the method of an object.
- "Using Function Notation with Methods" on page 13-18 — Shows you how to use standard function notation to call the methods of objects.

Naming Conventions for Properties and Methods

By convention, all properties begin with a capital letter, for example, the property `Name`. However, if a property consists of concatenated words, the words following the first word are capitalized, for example, the property `LabelString`. The same naming convention applies to methods, with the exception that a method name must begin with a letter in lowercase; for example, the method `findShallow`.

Using Dot Notation with Properties and Methods

You can access the properties and methods of an object by adding a period (`.`) and the name of the property or method to the end of an object's handle variable. For example, the following command returns the Type property for a State object represented by the handle `s`:

```
stype = s.Type
```

The following command calls the `dialog` method of the State object `s` to open a properties dialog for that state:

```
s.dialog
```

Nesting Dot Notation

You can nest smaller dot expressions in larger dot expressions of properties. For example, the `Chart` property of a State object returns the Chart object of the containing chart. Therefore, the expression `s.Chart.Name` returns the name of the chart containing the State whose object is `s`.

Methods can also be nested in dot expressions. For example, if the State object `sA1` represents state A1 in the final Stateflow chart at the end of "Create New Objects in the Chart" on page 13-11, the following command returns the string label for state A1's inner transition to its state A11.

```
label = sA1.innerTransitionsOf.LabelString
```

The preceding command uses the `LabelString` property of a Transition object and the `innerTransitions` method for a State object. It works as shown only because state A1 has one inner transition. If state A1 has more than one transition, you must first find all the inner transitions and then use an array index to access each one, as shown below:

```
innerTransitions = sA1.innerTransitionsOf
label1 = innerTransitions(1).LabelString
label2 = innerTransitions(2).LabelString
and so on...
```

Using Function Notation with Methods

As an alternative to dot notation, you can access object methods with standard function call notation. For example, you can use the `get` method to access the Name property of a Chart object, `ch`, through one of the following commands:

```
name = ch.get('Name')
name = get(ch,'Name')
```

If you have array arguments to methods you call, use function notation. The following example returns a vector of strings with the names of each chart in the array of Chart objects `chartArray`:

```
names = get(chartArray, 'Name')
```

If, instead, you attempt to use the `get` command with the following dot notation, an error results:

```
names = chartArray.get('Name')
```

Displaying Properties and Methods

The Stateflow API provides a few methods that help you see the properties and methods available for each object. These are described in the following sections.

- "Displaying the Names of Properties" on page 13-19 — Shows you how to display a list of the properties for a given object.
- "Displaying the Names of Methods" on page 13-20 — Shows you how to display a list of the methods for a given object.
- "Displaying Property Subproperties" on page 13-20 — Shows you how to display the subproperties of some properties.
- "Displaying Enumerated Values for Properties" on page 13-21 — Shows you how to display lists of acceptable values for properties that require enumerated values.

Displaying the Names of Properties

To access the names of all properties for any particular object, use the `get` method. For example, if the object `s` is a State object, enter the following command to list the properties and current values for any State object:

```
get(s)
```

Use a combination of the `get` method and the `classhandle` method to list only the names of the properties of an object. For example, list just the names of the properties for the state object `s` with the following command:

```
get(s.classhandle.Properties, 'Name')
```

To get a quick description for each property, use the `help` method. For example, if `s` is a State object, the following command returns a list of State object properties, each with a small accompanying description:

```
s.help
```

Note Some properties do not have a description, because their names are considered descriptive enough.

Displaying the Names of Methods

Use the `methods` method to list the methods for any object. For example, if the object `t` is a handle to a Transition object, use the following command to list the methods for any Transition object:

```
t.methods
```

Note The following internal methods might be displayed by the `methods` method for an object, but is not applicable to Stateflow use, and is not documented: `areChildrenOrdered`, `getChildren`, `getDialogInterface`, `getDialogSchema`, `getDisplayClass`, `getDisplayIcon`, `getDisplayLabel`, `getFullName`, `getHierarchicalChildren`, `getPreferredProperties`, `isHierarchical`, `isLibrary`, `isLinked`, `isMasked`.

Use a combination of the `get` method and the `classhandle` method to list only the names of the methods for an object. For example, list the names of the methods for the Transition object `t` with the following command:

```
get(t.classhandle.Methods, 'Name')
```

Displaying Property Subproperties

Some properties are objects that have properties referred to as subproperties. For example, when you invoke the command `get(ch)` on a chart object, `ch`, the output displays the following for the `StateFont` property:

```
StateFont: [1x1 Font]
```

This value indicates that the `StateFont` property of a state has subproperties. To view the subproperties of `StateFont`, enter the command `get(ch.StateFont.get)` to receive something like the following:

```
Name: Helvetica'
Size: 12
Weight: 'NORMAL'
Angle: 'NORMAL'
```

From this list it is clearly seen that `Name`, `Size`, `Weight`, and `Angle` are subproperties of the property `StateFont`. In the API property references for this guide (see “API References to Properties and Methods” on page 13-7),

these properties are listed by their full names: `Statefont.Name`, `Statefont.Size`, and so on.

Displaying Enumerated Values for Properties

Many of the properties for API objects can only be set to one of a group of enumerated strings. You can identify these properties from the API references for properties and methods (see "API References to Properties and Methods" on page 13-7). Generally, in the display for properties generated by the `get` command (see "Displaying the Names of Properties" on page 13-19) the values for these properties appear as strings of capital letters.

You display a list of acceptable strings for a property requiring enumerated values using the `set` method. For example, if `ch` is a handle to a Chart object, you can display the allowed enumerated values for the `Decomposition` property of that chart with the following command:

```
set (ch,'Decomposition')
```

Creating and Destroying API Objects

You create (construct), parent (contain), and delete (destroy) objects in Stateflow through constructor methods in the Stateflow API. For all but the Editor and Clipboard objects, creating objects establishes a handle to them that you can use for accessing their properties and methods to make modifications to Stateflow diagrams. See the following sections:

- "Creating Stateflow Objects" on page 13-22 — Shows you how to create and connect to Stateflow objects in the Stateflow API.
- "Establishing an Object's Parent (Container)" on page 13-24 — Shows you how to control the containment of graphical objects in the Stateflow diagram editor.

  Stateflow objects are contained (parented) by other objects as defined in the Stateflow hierarchy of objects (see "Stateflow API Object Hierarchy" on page 13-4). You control containment of nongraphical objects in the Stateflow Explorer.
- "Destroying Stateflow Objects" on page 13-25 — Shows you how to delete objects in the Stateflow API.

Creating Stateflow Objects

You create a Stateflow object as the child of a parent object through API constructor methods. Each Stateflow object type has its own constructor method. See "Constructor Methods" on page 15-5 for a list of the valid constructor methods.

Use the following process to create Stateflow objects with the Stateflow API:

1 Access the parent object to obtain a handle to it.

  When you first begin populating a model or chart, this means that you must get a handle to the Stateflow machine object or a particular chart object. See "Access the Machine Object" on page 13-9 and "Access the Chart Object" on page 13-10.

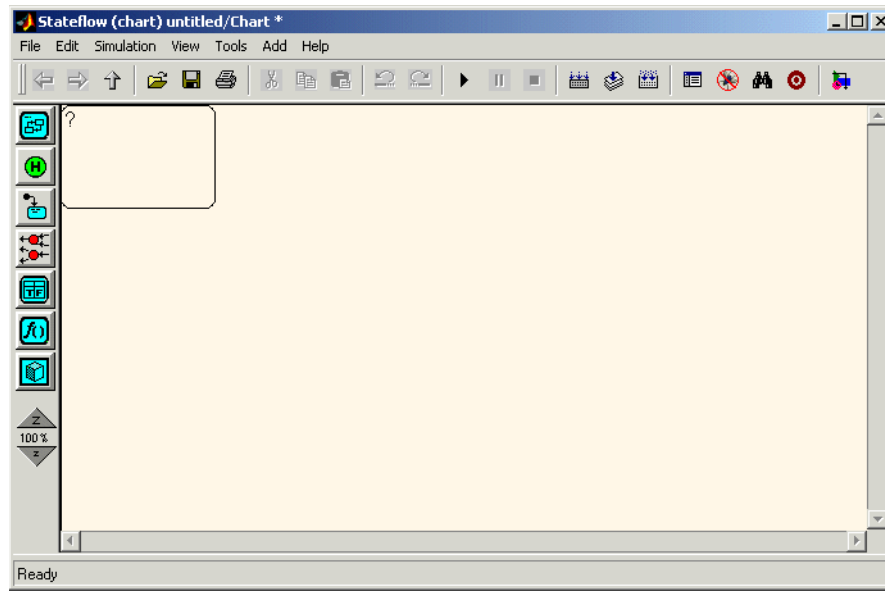
  See also "Accessing Existing Stateflow Objects" on page 13-26 for a more general means of accessing (getting an object handle to) an existing Stateflow object.

2 Call the appropriate constructor method for the creation of the object using the parent (containing) object as an argument.

For example, the following command creates and returns a handle `s` to a new state object in the chart object with the handle `ch`:

```
s = Stateflow.State(ch)
```

By default, the newly created state from the preceding command appears in the upper left-hand corner of the Stateflow chart (at *x*-*y* coordinates 0,0).



The constructor returns a handle to an API object for the newly created Stateflow object. Use this handle to display or change the object through its properties and methods.

3 Use the object handle returned by the constructor to make changes to the object in Stateflow.

For example, you can now use the handle `s` to set its name (`Name` property) and position (`Position` property). You can also connect it to other states or junctions by creating a Transition object and setting its `Source` or

`Destination` property to `s`. See "Create New Objects in the Chart" on page 13-11 for examples.

Use the preceding process to create all Stateflow objects in your chart. The section "Create New Objects in the Chart" on page 13-11 gives examples for creating states and transitions. Objects of other types are created just as easily. For example, the following command creates and returns a handle (`d1`) for a new Data object belonging to the state A (handle `sA`):

```
d1 = Stateflow.Data(sA)
```

Note Currently, there is no constructor for a Stateflow chart. To create a chart with the Stateflow API you must use the `sfnew` objectless method. See "Objectless Methods" on page 15-4 for a description.

Establishing an Object's Parent (Container)

As discussed in the previous section, "Creating Stateflow Objects" on page 13-22, the Stateflow API constructor establishes the parent for a newly created object by taking a handle for the parent object as an argument to the constructor.

Graphical Object Parentage

When graphical objects (states, boxes, notes, functions, transitions, junctions) are created, they appear completely inside their containing parent object. In the diagram editor, graphical containment is a necessary and sufficient condition for establishing the containing parent.

Repositioning a graphical object through its `Position` property can change an object's parent or cause an undefined parent error condition. Parsing a chart in which the edges of one object overlap with another produces an undefined parent error condition that cannot be resolved by the Stateflow parser. You can check for this condition by examining the value of the `BadIntersection` property of a Chart object, which equals 1 if the edges of a graphical object overlap with other objects. You need to set the size and position of objects so that they are clearly positioned and separate from other objects.

Nongraphical Object Parentage

When nongraphical objects (data, events, and targets) are created, they appear in the Stateflow Explorer at the hierarchical level of their owning object. Containment for nongraphical objects is established through the Stateflow Explorer only. See the section "The Stateflow Explorer Tool" on page 10-3.

Destroying Stateflow Objects

Each Stateflow object of type State, Box, Function, Note, Transition, Junction, Event, Data, or Target, has a destructor method named `delete`. In the following example, a State object, `s`, is deleted:

```
s.delete
```

The preceding command is equivalent to performing a mouse select and keyboard delete operation in the Stateflow diagram editor. Upon deletion, graphical Stateflow objects are sent to the Clipboard; nongraphical objects, such as data and events, are completely deleted. The workspace variable s still exists but is no longer a handle to the deleted state.

Accessing Existing Stateflow Objects

Creating Stateflow objects through the Stateflow API gives you an immediate handle to the newly created objects (see "Creating Stateflow Objects" on page 13-22). You can also connect to Stateflow objects that already exist for which you have no current API handle.

The following sections describe how you use the Stateflow API to find and access existing objects in Stateflow charts:

- "Finding Objects" on page 13-26 — Shows you how to use the `find` method, which is the most powerful and versatile method for locating objects in the Stateflow API.
- "Finding Objects at Different Levels of Containment" on page 13-27 — Shows you how to use the `find`, `findDeep`, and `findShallow` methods for finding objects at different levels of containment.
- "Getting and Setting the Properties of Objects" on page 13-28 — Shows you how to use the Stateflow API to access the properties of the existing objects you find in Stateflow charts.

Finding Objects

There are several object methods that you use to traverse the Stateflow hierarchy to locate existing objects. Chief among these is the versatile `find` method.

With the `find` method, you specify what to search for by specifying combinations of the following types of information:

- The type of object to find
- A property name for the object to find and its value

The following example searches through machine (model) `m` to return every State object with the name 'On'.

```
onState = m.find('-isa','Stateflow.State','-and','Name','On')
```

If a `find` command finds more than one object that meets its specifications, it returns an array of qualifying objects. The following example returns an array of all charts in your model:

```
chartArray = m.find('-isa','Stateflow.Chart')
```

Use array indexing to access individual properties and methods for a chart. For example, if the preceding command returns three Stateflow charts, the following command returns the Name property of the second chart found:

```
name2 = chartArray(2).Name
```

By default, the `find` command finds objects at all depths of containment within an object. This includes the zeroth level of containment, which is the searched object itself. For example, if state `A`, which is represented by State object `sA`, contains two states, `A1` and `A2`, and you specify a `find` command that finds all the states in `A` as follows,

```
states= sA.find( '-isa','Stateflow.State')
```

The preceding command finds three states: `A`, `A1`, and `A2`.

Note Be careful when specifying the objects you want to find with the `find` method for a Root or Machine object. Using the `find` method for these objects can return Simulink objects matching the arguments you specify. For example, if `rt` is a handle to the Root object, the command `find('Name', 'ABC')` might return a Simulink subsystem or block named `ABC`. See the reference for the `find` method for a full description of the method and its parameters.

Finding Objects at Different Levels of Containment

The `find` method finds objects at the depth of containment within an object that you specify. If you want to limit the containment search depth with the `find` command, use the `depth` switch. For example, to find all the objects in State object `sA` at the first level of containment, use the following command:

```
objArray = sA.find('-depth', 1)
```

Don't forget, however, that the `find` command always includes the zeroth level of containment, the containing object itself. So, the preceding command also includes state A in the list of objects found. See the reference for the `find` method in the "API Methods Reference".

The `findDeep` and `findShallow` methods are special modifications of the `find` method to find objects at particular depths of containment. For example, the

following command returns a collection of all junctions at the first level of containment inside the state A that is represented by State object `sA`:

```
juncArray = sA.findShallow('Junction')
```

The following command returns an array of all transitions inside state A at all levels of containment:

```
transArray = sA.findDeep('Transition')
```

You must always specify an object type when you use the `findShallow` and `findDeep` methods.

Getting and Setting the Properties of Objects

Once you obtain a particular object, you can access its properties directly or through the `get` method. For example, you obtain the description for a State object `s` with one of the following commands:

- `od = s.Description`
- `od = s.get ('Description')`
- `od = get (s, 'Description')`

You change the properties of an object directly or through the `set` method. For example, you change the description of the State object `s` with one of the following commands:

- `s.Description = 'This is the On state.'`
- `s.set ('Description', 'This is the On state.')`
- `set (s, 'Description', 'This is the On state.')`

Copying Objects

You can use the clipboard (accessed through the Stateflow API Clipboard object) to copy one object to another. See the following topics to learn how to copy objects from one object to another using the Clipboard object and its methods:

- "Accessing the Clipboard Object" on page 13-29 — Shows you how to use the Clipboard object and its two methods, `copy` and `pasteTo`, to copy objects from one object to another.
- "copy Method Limitations" on page 13-30 — Describes the features and limitations of the `copy` method that determine how you can copy objects in the Stateflow API.
- "Copying by Grouping (Recommended)" on page 13-30 — Gives you the most powerful method for copying using the `IsGrouped` property of State objects.
- "Copying Objects Individually" on page 13-31 — Tells you how you can also copy objects individually from one object to another.

Accessing the Clipboard Object

The Clipboard object (there is only one) provides an interface to the clipboard used in copying Stateflow objects. You cannot directly create or destroy the Clipboard object as you do other Stateflow API objects. However, you can attach a handle to it to use its properties and methods to copy Stateflow objects.

You create a handle to the Clipboard object by using the `sfclipboard` method as follows:

```
cb = sfclipboard
```

You create a handle to the Clipboard object (there is only one) with the following command:

```
cb = sfclipboard
```

Clipboard objects have two methods, `copy` and `pasteTo`, that together provide the functionality to copy objects from one object to another. The `copy` method copies the specified objects to the Clipboard object, and the `pasteTo` method pastes the contents of the Clipboard to a new container.

copy Method Limitations

The copy method is subject to the following limitations for all objects:

- The objects copied must be either *all* graphical (states, boxes, functions, transitions, junctions) or *all* nongraphical (data, events, targets).

  You cannot copy a mixture of graphical and nongraphical objects to the Clipboard in the same copy operation.
- To maintain the transition connections and containment relationships between copied objects, you must copy the entire array of related objects.

  All related objects must be part of the array of objects copied to the Clipboard. For example, if you attempt to copy two states connected by a transition to another container, you can only accomplish this by copying both the states and the transition at the same time. That is, you must do a single copy of a single array containing both the states and the transition that connects them.

  If you copy a grouped state to the Clipboard, not only are all the objects contained in the state copied, but all the relations among the objects in the grouped state are copied as well. Thus, copying by grouping is a recommended procedure. See “Copying by Grouping (Recommended)” on page 13-30.

Copying Graphical Objects

The copy method is subject to the following limitations for all graphical objects:

- Copying graphical objects also copies the Data, Event, and Target objects that the graphical objects contain.
- If all copied objects are graphical, they must all be seen in the same subviewer.

  This means that all graphical objects copied in a single copy command must reside in the same chart or subchart.

Copying by Grouping (Recommended)

Copying a grouped state in Stateflow copies not only the state but all of its contents. By grouping a state before you copy it, you can copy it and all of its contained objects at all levels of containment with the Stateflow API. This is the simplest way of copying objects and should be used whenever possible.

You use the boolean `IsGrouped` property for a state to group that state. If you set the `IsGrouped` property for a state to a value of true (=1), it is grouped. If you set `IsGrouped` to a value of false (=0), the state is not grouped.

The following example procedure copies state A to the chart X through grouping. In this example, assume that you already have a handle to state A and chart X through the MATLAB variables `sA` and `chX`, respectively.

1. If the state to copy is not already grouped, group it along with all its contents by setting the `IsGrouped` property for that state to true (=1).

```
prevGrouping = sA.IsGrouped
if (prevGrouping == 0)
   sA.IsGrouped = 1
end
```

2. Get a handle to the Clipboard object.

```
cb = sfclipboard
```

3. Copy the grouped state to the clipboard using the Clipboard object.

```
cb.copy(sA)
```

4. Paste the grouped object to its new container.

```
cb.pasteTo(chX)
```

5. Set the copied state and its source state to its previous `IsGrouped` property value.

```
sA.IsGrouped = prevGrouping
sNew = chX.find('-isa',Stateflow.State','-and','Name',sA.Name)
sNew.IsGrouped = prevGrouping
```

Copying Objects Individually

You can copy specific objects from one object to another. However, in order to preserve transition connections and containment relations between objects, you must copy all the connected objects at once. To accomplish this, use the general technique of appending objects from successive finds in MATLAB to a growing array of objects before copying the finished object array to the Clipboard.

Using the example of the Stateflow chart at the end of "Create New Objects in the Chart" on page 13-11, you can copy states A1, A2, and the transition connecting them to another state, B, with API commands. Assume that `sA` and `sB` are workspace handles to states A and B, respectively.

```
objArrayS = sA.findShallow('State')
objArrayT = sA.findShallow('Transition')
sourceObjs = {objArrayS ; objArrayT}
cb = sfclipboard
cb.copy(sourceObjs)
destObjs = cb.pasteTo(sB)
```

You can also accomplish the job of constructing the copy array through a complex `find` command. This might be adequate in certain situations. However, this approach might require you to formulate a very complex command. By contrast, the technique of appending found objects to an array relies on simpler find commands.

You can also copy nongraphical data, events, and target objects individually. However, since there is no way for these objects to find their new owners, you must ensure that each of these objects is separately copied to its appropriate owner object.

Note Copying objects individually is more difficult than copying grouped objects. This is why copying objects by grouping is recommended. See "Copying by Grouping (Recommended)" on page 13-30.

Using the Editor Object

The Editor object provides access to the purely graphical properties and methods of Chart objects. Each Chart object has its own Editor object. See the following topics to learn how to use the Editor object and its methods to change the display of the Stateflow diagram editor for a chart:

- "Accessing the Editor Object" on page 13-33 — Tells you how to connect to the Editor object.
- "Changing the Stateflow Display" on page 13-33 — Teaches you how to use the methods of the Editor object to change the display of the Stateflow diagram editor.

Accessing the Editor Object

You cannot directly create or destroy the Editor and Clipboard objects as you do other Stateflow API objects. However, you can attach a handle to them to use their properties and methods for modifications to Stateflow diagrams.

When you create a chart, an Editor object is automatically created for it. If `ch` is a workspace handle to a chart, you create a handle to the Editor object for that chart with the following command:

```
ed = ch.Editor
```

Changing the Stateflow Display

Use the handle `ed` from the preceding example to access the Editor object properties and methods. For example, the following command calls the `zoomIn` method to zoom in the chart by a factor of 20%:

```
ed.zoomIn
```

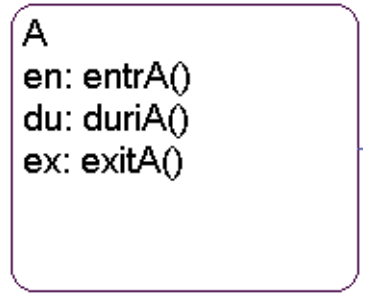
Or, you can simply set the `ZoomFactor` property of this chart's editor to an absolute zoom factor of 150%:

```
ed.ZoomFactor = 1.5
```

You can also use a chart's Editor object to change the window position of the diagram editor. For a reference to all the Editor object's properties and methods, see "Editor Properties" on page 15-6 and "Editor Methods" on page 15-7.

Entering Multiline Labels

In the examples shown thus far of entering labels for states and transitions, only a simple one-line expression has been used. The following figure shows state A with a multiline label.



There are two ways to enter multiline labels for both states and transitions. In the following examples, `sA` is a workspace variable handle to the State object in the Stateflow API for state A.

- Use the MATLAB function `sprintf`:

  ```
  str = sprintf('A\nen: entrA()\ndu: duriA()\nex: exitA()')
  sA.LabelString = str
  ```

  In this example, carriage returns are inserted into a string expression with the escape sequence `\n`.

- Use a concatenated string expression:

  ```
  str = ['A',10,'entr: entrA()',10,'du: duriA()',10,'ex: exitA()']
  sA.LabelString = str
  ```

  In this example, carriage returns are inserted into a concatenated string expression with the ASCII equivalent of a carriage return, the integer 10.

Creating Default Transitions

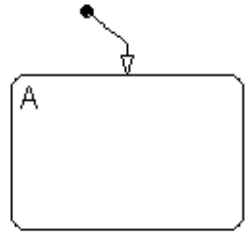
Default transitions differ from normal transitions in not having a source object. You can create a default transition with the following process:

1. Create a transition.
2. Attach the destination end of the transition to an object.
3. Position the source endpoint for the transition.

If you assume that the workspace variable sA is a handle to state A, the following commands create a default transition and position its source 25 pixels above and 15 pixels to the left of the top midpoint of state A:

```
dt = Stateflow.Transition(sA)
dt.Destination = sA
dt.DestinationOClock = 0
xsource = sA.Position(1)+sA.Position(3)/2-15
ysource = sA.Position(2)-25
dt.SourceEndPoint = [xsource ysource]
```

The created default transition has the following appearance:

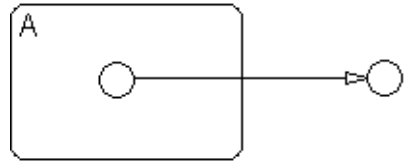


This method is also used for adding the default transitions toward the end of the example Stateflow diagram constructed in “Create New Objects in the Chart” on page 13-11.

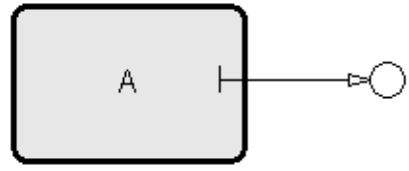
Making Supertransitions

The Stateflow API does not currently support the direct creation of supertransitions. Supertransitions are transitions between a state or junction in a top-level chart and a state or junction in one of its subcharts, or between states residing in different subcharts at the same or different levels in a diagram. For a better understanding of supertransitions, see “Using Supertransitions in Stateflow Charts” on page 5-71.

Stateflow does provide a workaround for indirectly creating supertransitions. In the following example, a supertransition is desired from a junction inside a subchart to a junction outside the subchart. In order to use the Stateflow API to create the supertransition in this example, first use the API to create the superstate as an ordinary state with a transition between its contained junction and a junction outside it.



Now set the `IsSubchart` property of the state A to true (=1).



This makes state `A` a subchart, and the transition between the junctions is now a supertransition.

You can also connect supertransitions to and from objects in an existing subchart (state `A`, for example) with the following procedure:

1. Save the original position of subchart `A` to a temporary workspace variable.

   For example, if the subchart `A` has the API handle `sA`, store its position with the following command:

```
sA_pos = sA.Position
```

2. Convert subchart `A` to a state by setting its `IsSubchart` property to false (=0).

   ```
   sA.IsSubchart = 0
   ```

3. Ungroup state `A` by setting its `IsGrouped` property to false (=0).

   ```
   sA.IsGrouped = 0
   ```

   When convert a subchart a normal state, it stays grouped to hide the contents of the subchart. When you ungroup the subchart, it might resize to display its contents.

4. Make the necessary transition connections.

   See "Create New Objects in the Chart" on page 13-11 for an example of creating a transition.

5. Set the `IsSubchart` property of state `A` back to true (=1).

   For example, `sA.IsSubchart = 1`

6. Assign subchart `A` its original position.

   ```
   sA.Position = sA_pos
   ```

   When you convert a subchart to a normal state and ungroup it, it might resize to accommodate the size of its contents. The first step of this procedure stores the original position of the subchart so that this can be restored after the transition connection is made.

Creating a MATLAB Script of API Commands

In the "Quick Start for the Stateflow API" section, you created and saved a new model through a series of Stateflow API commands. You can include the same API commands in the following MATLAB script, which allows you to quickly recreate the same model with a single command (`>> makeMyModel`).

```
function makeMyModel

sfnew;
m = sfroot('-isa','Stateflow.Machine');
chart = m.find('-isa', 'Stateflow.Chart');
sA = Stateflow.State(chart);
sA.Name = 'A';
sA.Position = [45 45 300 150];
sA1 = Stateflow.State(chart);
sA1.Name = 'A1';
sA1.Position = [80 80 90 80];
sA2 = Stateflow.State(chart);
sA2.Name = 'A2';
sA2.Position = [220 80 90 80];
tA1A2 = Stateflow.Transition(chart);
tA1A2.Source = sA1;
tA1A2.Destination = sA2;
tA1A2.SourceOClock = 3.;
tA1A2.DestinationOClock = 9.;
sA11 = Stateflow.State(chart);
sA11.Name = 'A11';
sA11.Position = [110 110 35 35];
tA1A11 = Stateflow.Transition(chart);
tA1A11.Source = sA1;
tA1A11.Destination = sA11;
tA1A11.SourceOClock = 1.;
tA1A11.DestinationOClock = 1.;
tA1A2.LabelString = 'E1';
tA1A11.LabelString = 'E2';
pos = tA1A2.LabelPosition;
pos(1) = pos(1)+15;
tA1A2.LabelPosition = pos;

dtA = Stateflow.Transition(chart)
```

```
dtA.Destination = sA
dtA.DestinationOClock = 0
xsource = sA.Position(1)+sA.Position(3)/2-10
ysource = sA.Position(2)-20
dtA.SourceEndPoint = [xsource ysource]
dtA1 = Stateflow.Transition(chart)
dtA1.Destination = sA1
dtA1.DestinationOClock = 0
xsource = sA1.Position(1)+sA1.Position(3)/2-10
ysource = sA1.Position(2)-20
dtA1.SourceEndPoint = [xsource ysource]
```

14

API Properties and Methods by Use

This reference section lists and categorizes the properties and methods of the Stateflow Application Programming Interface (API) by different types of use in Stateflow.

Reference Table Column Descriptions (p. 14-2)	Describes the columns appearing in the tables listing the properties and methods by use in the Stateflow API.
Categories of Use (p. 14-3)	Describes the categories of use that define each table listing of properties and methods in the Stateflow API.
Structural Properties (p. 14-5) Structural Methods (p. 14-12)	Describe the API properties and methods that directly or indirectly affect or reflect the hierarchical structure of Stateflow objects within their charts.
Behavioral Properties (p. 14-13) Behavioral Methods (p. 14-19)	Describe the properties and methods that affect the behavioral aspects of the Stateflow chart. They change how the chart behaves in the execution of its diagram.
Deployment Properties (p. 14-20) Deployment Methods (p. 14-25)	Describes the properties and methods that affect how code is generated and deployed from a Stateflow chart.
Utility and Convenience Properties (p. 14-26) Utility and Convenience Methods (p. 14-28)	Describes the properties and methods that maintain the infrastructure of the Stateflow API and also provide useful intermediate services for using the Stateflow API.
Graphical Properties (p. 14-30) Graphical Methods (p. 14-36)	Describes the properties and methods that perform services that are presentational (graphic) only and do not affect the actual Stateflow structure of objects.

Reference Table Column Descriptions

Reference tables for Stateflow API properties and methods have the following columns:

- **Name** — The name for the property or method. Each property or method has a name that you use in dot notation along with a Stateflow object to set or obtain the property's value or call the method.
- **Type** — A data type for the property. Some types are other Stateflow API objects, such as the `Machine` property, which is the Machine object that contains this object.
- **Access** — An access type for the property. Properties that are listed as RW (read/write) can be read and changed. For example, the `Name` and `Description` properties of particular objects are RW. However, some properties are RO (read-only) because they are set by MATLAB itself. For example, the `Dirty` property of Chart and Machine objects, which indicates whether a model has been changed in memory, is managed by the system and cannot be set directly by users.
- **Description** — A description for the property or method. For some properties, the equivalent GUI operations for setting it in Stateflow are also given.
- **Objects** — The types of objects that have this property or method. The object types are listed by a single letter corresponding to the beginning character of each object type (except for the Target object), which are as follows: Root (R), Machine (M), Chart (C), State (S), Box (B), Function (F), Truth Table (TT), Note (N), Transition (T), Junction (J), Event (E), Data (D), Target (X), Editor (ED), and Clipboard (CB).

Categories of Use

The following table lists and describes the use categories used to categorize the properties and methods of the Stateflow API in this reference:

Use	Description
Structural	Affect the hierarchical structure of Stateflow objects within their charts. See "Structural Properties" on page 14-5 and "Structural Methods" on page 14-12
Behavioral	Affect the behavioral aspects of the Stateflow chart. They change how the chart behaves in the execution of its diagram. See "Behavioral Properties" on page 14-13 and "Behavioral Methods" on page 14-19
Deployment	Affect how code is generated and deployed from a Stateflow chart. See "Deployment Properties" on page 14-20 and "Deployment Methods" on page 14-25
Utility and Convenience	Provide extra utility and convenience in manipulating Stateflow objects. See "Utility and Convenience Properties" on page 14-26 and "Utility and Convenience Methods" on page 14-28
Graphical	Affect only the appearance of Stateflow charts, leaving the underlying hierarchy of objects unaffected. See "Graphical Properties" on page 14-30 and "Graphical Methods" on page 14-36

Note Properties and methods listed in one use category can overlap into other use categories.

The properties and methods of the Stateflow API are also listed by their Stateflow objects. See “API Properties and Methods by Object” on page 15-1.

Structural Properties

Structural properties directly or indirectly affect or reflect the hierarchical structure of Stateflow objects within their charts.

See also “Structural Methods” on page 14-12.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Property	Type	Acc	Description	Objects
`BadIntersection`	Boolean	RO	If true, this object graphically intersects another state, box, or function in an invalid way.	S B F TT
`Chart`	Chart	RO	Chart object containing this object.	S B F N T J TT
`Decomposition`	Enum	RW	Set this property to `'EXCLUSIVE_OR'` to specify exclusive (OR) decomposition for the states at the first level of containment in this chart or state. Set to `'PARALLEL_AND'` to specify parallel (AND) decomposition for these states. Equivalent to the **Decomposition** selection in the context menu for this chart or state.	C S
`Destination`	State or Junction	RW	Destination state or junction of this transition. Assign `Destination` the destination object for this transition. You can also use the property `Destination` to detach the destination end of a transition through the command `t.Destination = []` where `t` is the Transition object.	T
`DestinationOClock`	Double	RW	Location of transition destination connection on state. Varies from 0 to 12 for full clock cycle location. Value taken as modulus 12 of entered value.	T

Property	Type	Acc	Description	Objects
`Iced`	Boolean	RO	Equivalent to property `Locked` except that this property is used internally to lock this object from being changed during activities such as simulation.	M C
`IsGrouped`	Boolean	RW	If set to true, group this state. Nothing is allowed to change inside a grouped state. This property is also useful for copying states to a new location. See "Copying by Grouping (Recommended)" on page 13-30.	S B F
`IsSubchart`	Boolean	RW	If set to true, makes this state, box, or function a subchart.	S B F
`LabelString`	String	RW	Label for this object. Equivalent to typing the label for this object in its label text field in the diagram editor.	S B F T TT
`Locked`	Boolean	RW	If set to true, prevents user from changing any Stateflow chart in this machine or chart.	M C
`Machine`	Machine	RO	Machine that contains this object.	C S B F N T J D E X TT
`MidPoint`	Rect	RW	Position of the midpoint of this transition relative to the upper left corner of the Stateflow diagram editor workspace, in a 1-by-2 (x,y) point array.	T
`Name`	String	RW	Name of this object. This property is RW except for the name of Machine object, which is RO.	M C S B F D E X TT
`ParsedInfo.Array.Size`	Integer	RO	Numeric equivalent of string Data property `Props.Array.Size`.	D

Property	Type	Acc	Description	Objects
`ParsedInfo.Array.FirstIndex`	Integer	RO	Numeric equivalent of string Data property `Props.Range.FirstIndex`.	D
`ParsedInfo.InitialValue`	Double	RO	Numeric equivalent of string Data property `Props.InitialValue`.	D
`ParsedInfo.Range.Maximum`	Double	RO	Numeric equivalent of string Data property `Props.Range.Maximum`.	D
`ParsedInfo.Range.Minimum`	Double	RO	Numeric equivalent of string Data property `Props.Range.Minimum`.	D
`Position`	Rect	RW	Position and size of this object's box in the Stateflow chart, given in the form of a 1-by-4 array consisting of the following: • (x,y) coordinates for the box's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the box	S B F N TT
`Position.Center`	Rect	RW	(x,y) position of junction relative to the upper left vertex of the parent chart or state.	J TT
`Position.Radius`	Double	RW	Radius of this junction.	J TT
`Props.Array.Size`	String	RW	Specifying a positive value for this property specifies that this data is an array of specified size. Equivalent to entering a positive value in the **Size** column for this data in the Explorer or in the **Sizes** field of the properties dialog for this data.	D

Property	Type	Acc	Description	Objects
`Props.Array.FirstIndex`	String	RW	Index of the first element of this data if it is an array (`Props.Array.Size >= 1`). Equivalent to entering a value of zero or greater in the **First Index** field of the Properties dialog for this data.	D
`Props.InitialValue`	String	RW	If the source of the initial value for this data is the Stateflow data dictionary, this is the value used. Equivalent to entering this value in the **InitVal** column for this data in the Explorer or similar field in the Properties dialog for this data.	D
`Props.Range.Maximum`	String	RW	Maximum value that this data can have during execution or simulation of the state machine. Equivalent to entering value in **Max** column for this data in Explorer or the **Max** field in the Properties dialog for this data.	D
`Props.Range.Minimum`	String	RW	Minimum value that this data can have during execution or simulation of the state machine. Equivalent to entering value in the **Min** column for this data in Explorer or in the **Min** field in the properties dialog for this data.	D

Property	Type	Acc	Description	Objects
`Scope`	Enum	RW	Scope of this data or event. Allowed values vary with the object containing this data or event and are as follows: The following apply to any data object: • `'LOCAL_DATA'` (Local) • `'CONSTANT_DATA'` (Constant) • `'OUTPUT_DATA'` (Output to Simulink) The following apply to data for machines only: • `'IMPORTED_DATA'` (Exported) • `'EXPORTED_DATA'` (Imported) The following apply to data for charts only: • `'INPUT_DATA'` (Input to Simulink) The following apply to data for charts and functions only: • `'TEMPORARY_DATA'` (Temporary) The following apply to data for functions only: • `'FUNCTION_INPUT_DATA'` (Input Data) • `'FUNCTION_OUTPUT_DATA'` (Output Data) The following apply to any event: • `'LOCAL_EVENT'` (Local) The following apply to events for charts only: • `'INPUT_EVENT'` (Input from Simulink) • `'OUTPUT_EVENT'` (Output to Simulink) The following apply to events for machines only: • `'IMPORTED_EVENT'` (Imported) • `'EXPORTED_EVENT'` (Exported)	D E

Property	Type	Acc	Description	Objects
`Source`	State or Junction	RW	Source state or junction of this transition. Assign `Source` the source object for this transition. You can also use the property `Source` to detach the source end of a transition with the command `t.Source = []` where `t` is the Transition object.	T
`SourceEndPoint`	Rect	RO*	x,y spatial coordinates for the endpoint of a transition. *This property can be changed only for default transitions. For all other transitions it is RO (read only).	T
`SourceOClock`	Double	RW	Location of transition source connection on state. Varies from 0 to 12 for full clock cycle location. Value taken as modulus 12 of entered value.	T
`Text`	String	RW	Label for this note. The text content for this note that you enter directly into the note in the diagram editor or in the **Label** field of the Properties dialog for this note.	N

Property	Type	Acc	Description	Objects
`Trigger`	Enum	RW	Type of signal that triggers this **Input to Simulink** or **Output to Simulink** event associated with its chart. Can be one of the following: • `'EITHER_EDGE_EVENT'` (Either Edge) • `'RISING_EDGE_EVENT'` (Rising Edge) • `'FALLING_EDGE_EVENT'` (Falling Edge) • `'FUNCTION_CALL_EVENT'` (Function Call) Equivalent to specifying the indicated parenthetical expression for the **Trigger** field of the Properties dialog for this event.	E
`Type`	Enum	RO	Type of this object. For states, can be one of the following: • `'OR'` (inclusive) • `'AND'` (parallel) The type of a state is determined by the parent's `Decomposition` property. For junctions, can be one of the following: • `'CONNECTIVE'` • `'HISTORY'`	S J

Structural Methods

Structural methods directly or indirectly affect the hierarchical structure of Stateflow objects within their charts.

See also “Structural Properties” on page 14-5.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Method	Description	Objects
`copy`	Copy the specified array of objects to the clipboard. See also `pasteTo` method.	CB
`delete`	Delete this object.	All but R M C CB ED
`pasteTo`	Paste the objects in the Clipboard to the specified container object. See also `copy` method.	CB
`set`	Set the specified property of this object with a specified value. Used with all objects except the Root object.	All
`Stateflow.Box`	Create a box for a parent chart, state, box, or function.	NA
`Stateflow.Data`	Create a data for a parent machine, chart, state, box, or function.	NA
`Stateflow.Event`	Create an event for a parent machine, chart, state, box, or function.	NA
`Stateflow.Function`	Create a graphical function for a parent chart, state, box, or function.	NA
`Stateflow.Junction`	Create a junction for a parent chart, state, box, or function.	NA
`Stateflow.Note`	Create a note for a parent chart or state.	NA
`Stateflow.State`	Create a state for a parent chart, state, box, or function.	NA
`Stateflow.Target`	Create a target for a parent machine.	NA

Behavioral Properties

Behavioral properties affect the behavioral aspects of the Stateflow chart. They change how the chart behaves in the execution of its diagram.

See also “Behavioral Methods” on page 14-19.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Property	Type	Acc	Description	Objects
`ActionTable`	Cell Array	RW	A cell array of strings containing the contents of the Action Table for this truth table.	TT
`ConditionTable`	Cell Array	RW	A cell array of strings containing the contents of the Action Table for this truth table.	TT
`Debug. Animation. Delay`	Double	RW	Specify a delay (slow down) value for animation. Equivalent to setting the **Delay (sec)** field in the Animation section of the Debugger window.	M
`Debug. Animation. Enabled`	Boolean	RW	If set to true, animation (simulation) is enabled. If false (=0), disabled. Equivalent to selecting the **Enabled** or **Disabled** radio button of the Animation section of the Debugger window.	M
`Debug. BreakOn. ChartEntry`	Boolean	RW	If set to true, set the chart entry breakpoint for all charts in this machine. Equivalent to selecting the **Chart Entry** check box on the Debugger window.	M
`Debug. BreakOn. EventBroadcast`	Boolean	RW	If set to true, set the event broadcast breakpoint for all charts in this machine. Equivalent to selecting the **Event Broadcast** check box on the Debugger window.	M

Property	Type	Acc	Description	Objects
`Debug.BreakOn.StateEntry`	Boolean	RW	If set to true, set the state entry breakpoint for all states in this machine. Equivalent to selecting the **State Entry** check box on the Debugger window.	M
`Debug.Breakpoints.EndBroadcast`	Boolean	RW	If set to true, set a debugger breakpoint for the end of the broadcast of this event. Equivalent to selecting the **End of broadcast** check box in the Properties dialog for this event.	E
`Debug.Breakpoints.StartBroadcast`	Boolean	RW	If set to true, set a debugger breakpoint for the start of the broadcast of this event. Equivalent to selecting the **Start of broadcast** check box in the properties dialog for this event.	E
`Debug.Breakpoints.onDuring`	Boolean	RW	If set to true, set the state during breakpoint for this chart. Equivalent to selecting the **State During** check box in the properties dialog for this state.	S
`Debug.Breakpoints.OnEntry`	Boolean	RW	If set to true, set the entry breakpoint for this object. Equivalent to selecting the **Chart Entry** or **State Entry** check box in the properties dialog for this chart or state, respectively.	C S
`Debug.Breakpoints.OnExit`	Boolean	RW	If set to true, set the state entry breakpoint for this object. Equivalent to selecting the **State Exit** check box in the properties dialog for this object.	S
`Debug.Breakpoints.WhenTested`	Boolean	RW	If set to true, set a debugging breakpoint to occur when this transition is tested to see if it is a valid transition. Equivalent to selecting the **When Tested** check box in the properties dialog of this transition.	T

Property	Type	Acc	Description	Objects
`Debug.Breakpoints.WhenValid`	Boolean	RW	If set to true, set a debugging breakpoint to occur when this transition has tested as valid. Equivalent to selecting the **When Valid** check box in the Properties dialog of this transition.	T
`Debug.DisableAllBreakpoints`	Boolean	RW	If set to true, disables the use of all breakpoints in this machine. Equivalent to selecting the **Disable all** check box in the Debugger window.	M
`Debug.RunTimeCheck.StateInconsistencies`	Boolean	RW	If set to true, check for state inconsistencies during a debug session. Equivalent to selecting the **State Inconsistency** check box in the Debugger window.	M
`Debug.RunTimeCheck.TransitionConflicts`	Boolean	RW	If set to true, check for transition conflicts during a debug session. Equivalent to selecting the **Transition Conflict** check box in the Debugger window.	M
`Debug.RunTimeCheck.CycleDetection`	Boolean	RW	If set to true, check for cyclical behavior errors during a debug session. Equivalent to selecting the **Detect Cycles** check box in the Debugger window.	M
`Debug.RunTimeCheck.DataRangeChecks`	Boolean	RW	If set to true, check for data range violations during a debug session. Equivalent to selecting the **Data Range** check box in the Debugger window.	M
`Debug.Watch`	Boolean	RW	If set to true, cause the debugger to halt execution if this data is modified. Setting this property to true is equivalent to selecting the **Watch** column entry for this data in the Explorer or selecting the **Watch in debug** check box in the Properties dialog for this data.	D

Property	Type	Acc	Description	Objects
`Decomposition`	Enum	RW	Set this property to `'EXCLUSIVE_OR'` to specify exclusive (OR) decomposition for the states at the first level of containment in this chart or state. Set to `'PARALLEL_AND'` to specify parallel (AND) decomposition for these states. Equivalent to selecting the **Decomposition** in the context menu for the chart or state.	C S
`EnableBitOps`	Boolean	RW	If set to true, enable C-like bit operations in generated code for this chart. Equivalent to selecting the **Enable C-like bit operations** check box in the chart Properties dialog.	C
`ExecuteAt Initialization`	Boolean	RW	If set to true, this chart's state configuration is initialized at time zero instead of first input event. Equivalent to selecting the **Execute (enter) Chart at Initialization** check box in the chart Properties dialog.	C
`LabelString`	String	RW	Label for this object. Equivalent to typing the label for this object in its label text field in the diagram editor.	S B F T
`SampleTime`	String	RW	Sample time for activating this chart. Applies only when the `UpdateMethod` property for this chart is set to `'DISCRETE'` (= **Sampled** in the **Update method** field in the Properties dialog for this chart).	C

Property	Type	Acc	Description	Objects
`Scope`	Enum	RW	Scope of this data or event. Allowed values vary with the object containing this data or event. The following apply to any data object: • `'LOCAL_DATA'` (Local) • `'CONSTANT_DATA'` (Constant) • `'OUTPUT_DATA'` (Output to Simulink) The following apply to data for machines: • `'IMPORTED_DATA'` (Exported) • `'EXPORTED_DATA'` (Imported) The following apply to data for charts: • `'INPUT_DATA'` (Input to Simulink) The following apply to data for charts and functions: • `'TEMPORARY_DATA'` (Temporary) The following apply to data for functions: • `'FUNCTION_INPUT_DATA'` (Input Data) • `'FUNCTION_OUTPUT_DATA'` (Output Data) The following apply to any event: • `'LOCAL_EVENT'` (Local) The following apply to events for charts: • `'INPUT_EVENT'` (Input from Simulink) • `'OUTPUT_EVENT'` (Output to Simulink) The following apply to events for machines: • `'IMPORTED_EVENT'` (Imported) • `'EXPORTED_EVENT'` (Exported) Above values are equivalent to entering the value shown in parentheses in the **Scope** field of the Properties dialog or the Explorer for this data.	D E

Property	Type	Acc	Description	Objects
`Trigger`	Enum	RW	Type of signal that triggers this Input to Simulink or Output to Simulink event associated with its chart. Can be `'EITHER_EDGE_EVENT'` or `'FUNCTION_CALL_EVENT'`. Equivalent to specifying **Either Edge** or **Function Call**, respectively, in the **Trigger** field of the Properties dialog for this event.	E
`UpdateMethod`	Enum	RW	Activation method of this chart. Can be `'INHERITED'`, `'DISCRETE'`, or `'CONTINUOUS'`. Equivalent to the **Update method** field in the Properties dialog for this chart, which takes one of the following corresponding selections: **Triggered or Inherited**, **Sampled**, **Continuous**.	C

Behavioral Methods

Behavioral methods affect the behavioral aspects of the Stateflow chart. They change how the chart behaves in the execution of its diagram.

See “Behavioral Properties” on page 14-13.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Method	Description	Objects
`outputData`	Output the activity status of this state to Simulink via a data output port on the Chart block of this state.	S
`parse`	Parses all the charts in this machine (model) or just this chart.	M C

Deployment Properties

Deployment properties affect how code is generated and deployed from a Stateflow chart.

See “Deployment Methods” on page 14-25.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Property	Type	Acc	Description	Objects
`ApplyToAllLibs`	Boolean	RW	If set to true, use settings in this target for all libraries. Equivalent to selecting the **Use settings for all libraries** check box in this target's Target Builder dialog.	X

`CodeFlagsInfo`	`Array`	`RO`	A MATLAB vector of structures containing information on `n` code flag settings for this target. Each element in the vector is a MATLAB structure with information about a particular code flag. Each flag corresponds to a selection in the **Coder Options** dialog for this target. If you want to see information about the first flag for a Target object `t`, use the following commands:
			`cfi = t.CodeFlagsInfo`
			`disp(cfi(1))`
			If you want to quickly see the names of all the flags, do the following:
			`cfi.name`
			The `Name` member of the `CodeFlagsInfo` structure is shorthand for a longer expression in the **Coder Options** dialog. For example, the name 'comments' actually refers to the dialog setting **Comments in generated code**.
			You use the name of a code flag to get and set the code flag value with the methods `getCodeFlag` and `setCodeFlag`. Changing the vector returned by `CodeFlagsInfo` does not change an actual flag.
				X

`CodegenDirectory`	String	RW	Directory to receive generated code. Applies only to targets other than `sfun` and `rtw` targets.	X

Property	Type	Acc	Description	Objects
`CustomCode`	String	RW	Custom code included at the top of the generated code. Equivalent to the entry for the **Custom code included at the top of generated code** selection of the **Target Options** dialog for this target.	X
`CustomInitializer`	String	RW	Custom initialization code. Equivalent to the entry for the **Custom initialization code (called from mdlInitialize)** selection of the **Target Options** dialog for this target. Applies only to `sfun` and `rtw` targets.	X
`CustomTerminator`	String	RW	Custom termination code. Equivalent to the entry for the **Custom termination code (called from mdlTerminate)** selection of the **Target Options** dialog for this target. Applies only to `sfun` and `rtw` targets.	X
`DataType`	Enum	RW	Data type of this data. Can have one of the following possible values: `'boolean'`, `'uint8'`, `'int8'`, `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'single'`, `'double'` and `'fixpt'`. Equivalent to an entry in the **Type** column for this data in Explorer or the **Type** field in the properties dialog for this data.	D
`EnableBitOps`	Boolean	RW	If set to true, enable C-like bit operations in generated code for this chart. Equivalent to selecting the **Enable C-like bit operations** check box in the chart properties dialog.	C
`ExecuteAt Initialization`	Boolean	RW	If set to true, initialize this chart's state configuration at time zero instead of first input event. Equivalent to selecting the **Execute (enter) Chart at Initialization** check box in the chart properties dialog.	C

Property	Type	Acc	Description	Objects
`ExportChart Functions`	Boolean	RW	If set to true, graphical functions at chart level are made global. Equivalent to selecting the **Export Chart Level Graphical Functions (Make Global)** check box in the chart properties dialog.	C
`FixptType. Bias`	Double	RW	The Bias value for this fixed-point type.	D
`FixptType. FractionalSlope`	Double	RW	The Fractional Slope value for this fixed-point type.	M
`FixptType. RadixPoint`	Integer	RW	The power of two specifying the binary point location for this fixed-point type.	D
`FixptType. BaseType`	Enum	RW	The size and sign of the base for the quantized integer, Q, of this fixed-point type.	D
`isLibrary`	Boolean	RO	If `true`, specifies that the current model builds a library and not an application.	M
`SaveToWorkspace`	Boolean	RW	If set to true, this data is saved to the MATLAB workspace. Equivalent to selecting the **ToWS** column entry for this data in the Explorer or selecting the **Save final value to base workspace** field in the properties dialog for this data.	D
`StrongDataTyping WithSimulink`	Boolean	RW	If set to true, set strong data typing with Simulink I/O. Equivalent to the **Use Strong Data Typing with Simulink I/O** check box in the chart properties dialog.	C
`UserIncludeDirs`	Boolean	RW	Custom include directory paths. Equivalent to the entry for the **Custom include directory paths** selection of the **Target Options** dialog for this target.	X

Property	Type	Acc	Description	Objects
`UserLibraries`	String	RW	Custom libraries. Equivalent to the entry for the **Custom libraries** selection of the **Target Options** dialog for this target.	X
`UserSources`	String	RW	Custom source files. Equivalent to the entry for the **Custom source files** selection of the **Target Options** dialog for this target.	X

Deployment Methods

Deployment methods affect how code is generated and deployed from a Stateflow chart.

See also “Deployment Properties” on page 14-20.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Method	Description	Objects
`build`	Build this target only for those portions of the target's charts that have changed since the last build (i.e., incrementally). See also the methods `rebuildAll`, `generate`, `rebuildAll`, and `make`.	X
`generate`	Generate code for this target only for those portions of this target's charts that have changed since the last code generation (i.e., incrementally). See also the methods `build`, `rebuildAll`, `regenerateAll`, and `make`.	X
`getCodeFlag`	Return the value of the specified code flag for this target.	X
`make`	Compile this target for only those portions of this target's charts that have changed since the last compile (i.e., incrementally). For a simulation target (sfun), a dynamic link library (`sfun.dll`) is compiled from the generated code. See also the methods `build`, `rebuildAll`, `generate`, and `regenerateAll`.	X
`rebuildAll`	Completely rebuild this target. See also the methods `build`, `generate`, `regenerateAll`, and `make`.	X
`regenerateAll`	Completely regenerate code for this target. See also the methods `build`, `rebuildAll`, `generate`, and `make`.	X
`setCodeFlag`	Set the value of the specified code flag for this target.	X

Utility and Convenience Properties

Utility and convenience properties maintain the infrastructure of the Stateflow API and also provide useful intermediate services for using the Stateflow API.

See “Utility and Convenience Methods” on page 14-28.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Property	Type	Acc	Description	Objects
`Created`	String	RO	Date of creation of this machine.	M
`Creator`	String	RW	Creator of this machine.	M
`Description`	String	RW	Description of this object. Equivalent to entering a description in the **Description** field of the Properties dialog for this object.	M C S B F N T J D E X TT
`Dirty`	Boolean	RO	If true, this object has changed since it was opened or saved.	M C
`Document`	String	RW	Document link to this note. Equivalent to entering the **Document Link** field of the Properties dialog for this object.	M C S B F N T J D E X TT
`FullFileName`	String	RO	Full path name of file under which this machine (model) is stored.	M
`Id`	Integer	RO	Unique identifier assigned to this object to distinguish it from other objects loaded in memory.	M C S B F N T J D E X TT
`Modified`	String	RW	Comment area for entering date and name of modification to this machine (model).	M

Property	Type	Acc	Description	Objects
`OverSpec Diagnostic`	String	RW	Interprets the error diagnosis of this truth table as overspecified according to the possible values 'Error', 'Warning', or 'None'. In the truth table editor, the value of this property is assigned by selecting **Overspecified** from the **Diagnostics** menu item and then selecting one of the three values.	TT
`SfVersion`	Double	RO	Full version number for current Stateflow. For example, the string `'41112101'` appears for Stateflow version 4.1.1 and MATLAB version 12.1. The remaining `'01'` is for internal use.	M
`Tag`	Any Type	RW	A field you can use to hold data of any type for this object.	M C S B F T J D E X TT
`UnderSpec Diagnostic`	String	RW	Interprets the error diagnosis of this truth table as underspecified according to the possible values 'Error', 'Warning', or 'None'. In the truth table editor, the value of this property is assigned by selecting **Underspecified** from the **Diagnostics** menu item and then selecting one of the three values.	TT
`Units`	String	RW	Physical units corresponding to the value of this data object.	D
`Version`	String	RW	Comment string for recording version of this model.	M

Utility and Convenience Methods

Utility and convenience properties maintain the infrastructure of the Stateflow API and also provide useful intermediate services for using the Stateflow API.

See also "Utility and Convenience Properties" on page 14-26.

For a key to the abbreviations for object types in the Object column, see "Reference Table Column Descriptions" on page 14-2.

Method	Description	Objects
`classhandle`	Return a read-only handle to the class (object type) used to define this object.	All
`defaultTransitions`	Return the default transitions in this chart at the top level of containment.	C S B F
`dialog`	Display the Properties dialog of this object.	M C S B F N T J D E X
`find`	Return objects of this object that meet the criteria specified by the arguments.	All
`findDeep`	Return all objects of the specified type in this object at all levels of containment (i.e., infinite depth).	R M C S B F
`findShallow`	Return all objects of the specified type in this object at only the first level of containment (i.e., first depth).	R M C S B F
`get`	Display the property settings of this object.	All
`help`	Display a list of properties for this object with short descriptions. Used with all objects except the Root and Machine object.	All
`innerTransitions`	Return the inner transitions that originate with this object and terminate on a contained object.	S B F
`methods`	Return the methods of this object.	All

Method	Description	Objects
`outerTransitions`	Return an array of transitions that exit the outer edge of this object and terminate on an object outside the containment of this object.	S B F
`sfclipboard`	Return a handle to the Clipboard object.	None
`sfexit`	Close all Stateflow diagrams and Simulink models containing Stateflow diagrams, and exit the Stateflow environment. **Caution** Executing the `sfexit` method causes all models in Simulink to close without a request to save.	None
`sfhelp`	Display Stateflow online help in the MATLAB Help browser.	None
`sfnew`	Create and display a new Simulink model containing an empty Stateflow block.	None
`sfprint`	Print the visible portion of a Stateflow diagram.	None
`sfroot`	Return a handle to the Root object.	None
`sfsave`	Save the current machine and Simulink model.	None
`sfversion`	Return the current version of Stateflow.	None
`stateflow`	Open the current Stateflow model window.	None
`sourcedTransitions`	Return all inner and outer transitions whose source is this object.	S B F J
`struct`	Return a MATLAB structure containing the property settings of this object.	C S B F N T J D E X
`view`	Display this object in a Stateflow diagram editor.	C S B F N T J D E X

Graphical Properties

Graphical properties perform services that are presentational (graphic) only and do not affect the actual Stateflow structure of objects. Those graphical properties that do affect this underlying structure are included in the section “Structural Properties” on page 14-5.

See also “Graphical Methods” on page 14-36.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Property	Type	Acc	Description	Objects
`Alignment`	Enum	RW	Alignment of text in note box. Can be `'LEFT'`, `'CENTER'`, or `'RIGHT'`.	N
`ArrowSize`	Double	RW	Size of transition arrows coming into this object. Equivalent to selecting **Arrowhead Size** from the context menu for this state.	S B F T J
`ChartColor`	[R,G,B]	RW	Background color of this chart in a 1-by-3 RGB array with each value normalized on a scale of 0 to 1.	C
`DrawStyle`	String	RW	Set the drawing style for this transition. Set to `'STATIC'` for static transitions or `'SMART'` for smart transitions. Equivalent to selecting the **Smart** switch toggle from the context menu for this transition.	T
`Editor`	Editor	RO	Editor object for this chart.	C
`ErrorColor`	[R,G,B]	RW	Set the RGB color for errors in the Stateflow Diagram Editor in a 1-by-3 RGB array with each value normalized on a scale of 0 to 1. Equivalent to changing **Error** color in the **Colors & Fonts** dialog under **Edit** > **Style**.	C

Property	Type	Acc	Description	Objects
`Font.Angle`	Enum	RW	Style of the font for the text in this note. Can be `'ITALIC'` or `'NORMAL'`. This property overrides the default style for this note, which is set by the `StateFont.Angle` property of the Chart object containing this note.	N
`Font.Name`	String	RO	Name of the font for the text in this note. This property is read-only (RO) and set by the `StateFont.Name` property of the Chart object containing this note.	N
`Font.Size`	Double	RW	Size of the font for the label text for this note. This property overrides the default size for this note, which is set by the `StateFont.Size` property of the Chart object containing this note. Equivalent to selecting **Font Size** > **<font size>** in the context menu for this note.	N
`Font.Weight`	Enum	RW	Weight of the font for the label text for this note. Can be `'BOLD'` or `'NORMAL'`. This property overrides the default weight for the text in this note, which is set by the `StateFont.Weight` property of the Chart object containing this note.	N
`FontSize`	Double	RW	Size of the font for the label text for this object. This property overrides the default size for this object, which is set by the `StateFont.Size` property (`TransitionFont.Size` for transitions) of the Chart object containing this object. Equivalent to selecting **Font Size** > **<font size>** in the context menu for this object.	S B F T TT
`Interpretation`	Enum	RW	How the text in this note is interpreted for text processing. Can be `'NORMAL'` or `'TEX'`.	N

Property	Type	Acc	Description	Objects
`JunctionColor`	[R,G,B]	RW	Set the RGB color for junctions in the Stateflow Diagram Editor in a 1-by-3 RGB array with each value normalized on a scale of 0 to 1. Equivalent to changing the **Junction** color in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`LabelPosition`	Rect	RW	Position and size of this transition's label in the Stateflow chart, given in the form of a 1-by-4 array consisting of the following: • (x,y) coordinates for the label's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the label	T
`SelectionColor`	[R,G,B]	RW	Color of selected items for this chart in a 1-by-3 RGB array with each value normalized on a scale of 0 to 1. Equivalent to changing the **Selection** color in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`StateColor`	[R,G,B]	RW	Color of the state box in a 1-by-3 RGB array with each value normalized on a scale of 0 to 1. Equivalent to changing the **State/Frame** color in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`StateFont.Angle`	Enum	RW	Font angle for the labels of State, Box, Function, and Note objects. Can be `'ITALIC'` or `'NORMAL'`. Equivalent to **Italic** and **Regular** settings when changing the font style of the **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with property `StateFont.Weight` to achieve Bold Italic style. You can individually override this property with the `Font.Angle` property for Note objects.	C

Property	Type	Acc	Description	Objects
`StateFont.Name`	String	RW	Font style used for the labels of State, Box, Function, and Note objects. Enter a string for the font name — no selectable values. Font remains set to previous font for unrecognized font strings. Equivalent to changing the font of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`StateFont.Size`	Integer	RW	Font size for the labels of State, Box, Function, and Note objects. Equivalent to changing the font size of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. You can individually override this property with the `FontSize` property for State, Box, and Function objects and with the `Font.Size` property for Note objects.	C
`StateFont.Weight`	Enum	RW	Font weight for state labels. Can be `'BOLD'` or `'NORMAL'`. Equivalent to **Bold** and **Regular** settings of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with the property `StateFont.Angle` to achieve Bold Italic style. You can individually override this property with the `Font.Weight` property for Note objects.	C
`StateLabelColor`	[R,G,B]	RW	Color of the state labels for this chart in 1-by-3 RGB array with each value normalized on a scale of 0 to 1. Equivalent to changing the label color of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`Subviewer`	Chart or State	RO	State or chart in which this object can be graphically viewed.	S B F N T J TT

Property	Type	Acc	Description	Objects
`TransitionColor`	[R,G,B]	RW	Set the RGB color for transitions in the Stateflow Diagram Editor in a 1-by-3 RGB array with each value normalized on a scale of 0 to 1. Equivalent to changing the **Transition** color in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`TransitionFont.Angle` (Enum, RW)	Enum	RW	Font angle for state labels. Can be `'ITALIC'` or `'NORMAL'`. Equivalent to **Italic** and **Regular** settings when changing font style of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with property `StateFont.Weight` to achieve Bold Italic style.	C
`TransitionFont.Name`	String	RW	Font used for transition labels. Enter string for font name (no selectable values). Font remains set to previous font for unrecognized font strings. Equivalent to changing the font of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`TransitionFont.Size`	Integer	RW	Default font size for transition labels. Truncated to closest whole number less than or equal to entered value. Equivalent to changing font size of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`TransitionFont.Weight`	Enum	RW	Font weight for transition labels. Can be `'BOLD'` or `'NORMAL'`. Equivalent to **Bold** and **Regular** settings when changing font style of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with property `StateFont.Angle` to achieve Bold Italic style.	C

Property	Type	Acc	Description	Objects
`TransitionLabelColor`	[R,G,B]	RW	Color of the transition labels for this chart in 1-by-3 RGB array with each value normalized on a scale of 0 to 1. Equivalent to changing the label color of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.	C
`Visible`	Boolean	RO	If true, indicates that this object is currently displayed in a Stateflow diagram editor window.	C
`WindowPosition`	Rect	RW	Position and size of this chart given in the form of a 1-by-4 array consisting of the following: • (x,y) coordinates for the window's left bottom vertex relative to the lower left corner of the screen • Width and height of the box	ED
`ZoomFactor`	Double	RW	View magnification level (zoom factor) of this chart in the chart diagram editor.	ED

Graphical Methods

Graphical methods perform services that are presentational (graphic) only and do not affect the actual Stateflow structure of objects.

See also “Graphical Properties” on page 14-30.

For a key to the abbreviations for object types in the Object column, see “Reference Table Column Descriptions” on page 14-2.

Method	Description	Objects
`zoomIn and zoomOut`	Causes the Stateflow chart editor to zoom in or zoom out on this chart.	ED

15

API Properties and Methods by Object

Reference Table Columns (p. 15-3)	Describes the columns appearing in the tables listing the properties and methods by object in the Stateflow API.
Objectless Methods (p. 15-4)	Describes API methods that do not belong to any object.
Constructor Methods (p. 15-5)	Lists the constructor methods for each Stateflow object creatable through a constructor.
Editor Properties (p. 15-6)	Descriptions of properties and methods for the Editor (diagram editor) in the Stateflow API.
Editor Methods (p. 15-7)	
Clipboard Methods (p. 15-8)	Descriptions of the methods of the Clipboard used for copying and pasting objects from chart to chart.
All Object Methods (p. 15-9)	Describes methods that belong to all Stateflow objects.
Root Methods (p. 15-10)	Description of the methods of the Root object that contains all other Stateflow objects
Machine Properties (p. 15-11)	Descriptions of properties and methods for the Machine object (model) in the Stateflow API.
Machine Methods (p. 15-14)	
Chart Properties (p. 15-16)	Descriptions of properties and methods for Chart objects (charts) in the Stateflow API.
Chart Methods (p. 15-22)	
State Properties (p. 15-24)	Descriptions of properties and methods for State objects (states) in the Stateflow API.
State Methods (p. 15-27)	
Box Properties (p. 15-29)	Descriptions of properties and methods for Box objects (boxes) in the Stateflow API.
Box Methods (p. 15-31)	
Function Properties (p. 15-33)	Descriptions of properties and methods for Function objects (graphical functions) in the Stateflow API.
Function Methods (p. 15-35)	

Truth Table Properties (p. 15-37) Truth Table Methods (p. 15-40)	Descriptions of properties and methods for Truth Table objects in the Stateflow API.
Note Properties (p. 15-41) Note Methods (p. 15-43)	Descriptions of properties and methods for Note objects (notes) in the Stateflow API.
Transition Properties (p. 15-44) Transition Methods (p. 15-47)	Descriptions of properties and methods for Transition objects (transitions) in the Stateflow API.
Junction Properties (p. 15-48) Junction Methods (p. 15-49)	Descriptions of properties and methods for Junction objects (junctions) in the Stateflow API.
Data Properties (p. 15-50) Data Methods (p. 15-55)	Descriptions of properties and methods for Data objects (data) in the Stateflow API.
Event Properties (p. 15-56) Event Methods (p. 15-58)	Descriptions of properties and methods for Event objects (events) in the Stateflow API.
Target Properties (p. 15-59) Target Methods (p. 15-63)	Descriptions of properties and methods for Target objects (targets) in the Stateflow API.

Reference Table Columns

Reference tables for Stateflow API properties and methods have the following columns:

- **Name** — The name for the property or method. Each property or method has a name that you use in dot notation along with a Stateflow object to set or obtain the property's value or call the method.
- **Type** — A data type for the property. Some types are other Stateflow API objects, such as the `Machine` property, which is the Machine object that contains this object.
- **Access** — An access type for the property. Properties that are listed as RW (read/write) can be read and changed. For example, the `Name` and `Description` properties of particular objects are RW. However, some properties are RO (read-only) because they are set by MATLAB itself. For example, the `Dirty` property of Chart and Machine objects, which indicates whether a model has been changed in memory, is managed by the system and cannot be set directly by users.
- **Description** — A description for the property or method. For some properties, the equivalent GUI operations in Stateflow for setting it are also given.

Objectless Methods

The following methods do not belong to any API object, including Stateflow objects. See “API Methods Reference” on page 16-1 for details on each method.

Description	Method
`sfclipboard`	Return a handle to the Clipboard object.
`sfexit`	Close all Stateflow diagrams and Simulink models containing Stateflow diagrams, and exit the Stateflow environment. **Caution** Executing the `sfexit()` method causes all models in Simulink to close without a request to save them.
`sfhelp`	Display Stateflow online help in the MATLAB Help browser.
`sfnew`	Create and display a new Simulink model containing an empty Stateflow block.
`sfprint`	Print the visible portion of a Stateflow diagram.
`sfroot`	Return a handle to the Root object.
`sfsave`	Save the current machine and Simulink model.
`sfversion`	Return the current version of Stateflow.
`stateflow`	Open the current Stateflow model window.

Constructor Methods

The following methods create a new Stateflow object for a parent object specified as an argument in the general expression `o = Stateflow.Object(p)`, where `o` is a handle to an API object for the new Stateflow object, `p` is a handle to the parent object, and *`Object`* is the type of the object:

Method	Description
`Stateflow.Box`	Create a box for a parent chart, state, box, or function.
`Stateflow.Data`	Create a data for a parent machine, chart, state, box, or function.
`Stateflow.Event`	Create an event for a parent machine, chart, state, box, or function.
`Stateflow.Function`	Create a graphical function for a parent chart, state, box, or function.
`Stateflow.Junction`	Create a junction for a parent chart, state, box, or function.
`Stateflow.Note`	Create a note for a parent chart, state, box, or function.
`Stateflow.State`	Create a state for a parent chart, state, box, or function.
`Stateflow.Target`	Create a target for a parent machine.

Editor Properties

The Editor object has the properties in the table below. See also "Editor Methods" on page 15-7.

Property	Type	Acc	Description
`WindowPosition`	Rect	RW	Position and size of this chart given in the form of a 1-by-4 array consisting of the following: • (x,y) coordinates for the window's left bottom vertex relative to the lower left corner of the screen • Width and height of the box Default value = `[124.3125 182.8125 417 348.75]`
`ZoomFactor`	Double	RW	View magnification level (zoom factor) of this chart in the chart diagram editor. A value of 1 corresponds to a zoom factor of 100%, 2 to a value of 200%, and so on. Default value = 1.

Editor Methods

The Editor object has the methods displayed in the table below. For details on each method, see the“API Methods Reference” on page 16-1.

See also “Editor Properties” on page 15-6.

Method Name	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this Editor object.
`disp`	Display the property names and their settings for this Editor object.
`get`	Return the specified property settings for the Editor object.
`help`	Display a list of properties for this Editor object with short descriptions.
`methods`	Display all nonglobal methods of this Editor object.
`set`	Set the specified property of this Editor object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Editor object.
`zoomIn and zoomOut`	Cause the Stateflow chart editor to zoom in or zoom out on this chart.

Clipboard Methods

The Clipboard object has the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

Method Name	Description
`classhandle`	Return a read-only handle to the class (object type) used to define the Clipboard object.
`copy`	Copy the objects specified to this Clipboard object.
`get`	Return the specified property settings for this Clipboard object.
`help`	Display a list of properties for this Clipboard object with short descriptions.
`methods`	Display all nonglobal methods of this Clipboard object.
`pasteTo`	Paste the contents of this clipboard to the specified container object.
`set`	Set the specified property of this Clipboard object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Clipboard object.

All Object Methods

The following methods apply to all API objects including those of Stateflow. Only object-exclusive methods appear when you use the method `methods` to display methods for an object. However, the tables of methods for each API object that follow do list these methods as if they were their own.

See the "API Methods Reference" on page 16-1 for details on each method.

Method Name	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this object.
`delete`	Delete this object. Used with all objects except the Root, Machine, Chart, Clipboard, and Editor objects.
`disp`	Display the property names and their settings for this object.
`find`	Find all objects of this object that meet the specified criteria.
`get`	Return the specified property settings for this object.
`methods`	Display all nonglobal methods of this object.
`set`	Set the specified property of this object with a specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this object.

Root Methods

The Root object has the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this Root object.
`find`	Find all objects that this Root object contains that meet the specified criteria.
`findDeep`	Return all objects of the specified type in this Root object at all levels of containment (i.e., infinite depth).
`findShallow`	Return all objects of the specified type in this Root object at only the first level of containment (i.e., first depth).
`get`	Return the specified property settings for the Root object.
`help`	Display a list of properties for the Root object with short descriptions.
`methods`	Display all nonglobal methods of this Root object.
`set`	Set the specified property of this Root object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Root object.

Machine Properties

Stateflow API objects of type Machine have the properties shown in the table below. See also “Machine Methods” on page 15-14.

Property	Type	Acc	Description
`Created`	String	RO	Date of creation of this machine.
`Creator`	String	RW	Creator (default = `'Unknown'`) of this machine.
`Debug.Animation.Enabled`	Boolean	RW	If set to `true` (default), animation (simulation) is enabled. If `false`, disabled. Equivalent to the **Enabled** or **Disabled** radio button of the Animation section of the Debugger window.
`Debug.Animation.Delay`	Double	RW	Specify a value to delay (slow down) animation (default value = `0`). Equivalent to the **Delay (sec)** field in the Animation section of the Debugger window.
`Debug.BreakOn.ChartEntry`	Boolean	RW	If set to `true` (default = `false`), set the chart entry breakpoint for all charts in this machine. Equivalent to the **Chart Entry** check box in the Debugger window.
`Debug.BreakOn.EventBroadcast`	Boolean	RW	If set to `true` (default = `false`), set the event broadcast breakpoint for all charts in this machine. Equivalent to the **Event Broadcast** check box in the Debugger window.
`Debug.BreakOn.StateEntry`	Boolean	RW	If set to `true` (default = `false`), set the state entry breakpoint for all charts in this machine. Equivalent to the **State Entry** check box in the Debugger window.
`Debug.DisableAllBreakpoints`	Boolean	RW	If set to `true` (default = `false`), disable the use of all breakpoints in this machine. Equivalent to the **Disable all** check box in the Debugger window.

Property	Type	Acc	Description
`Debug.RunTimeCheck.CycleDetection`	Boolean	RW	If set to `true`, check for cyclical behavior errors during a debug session. Equivalent to the **Detect Cycles** check box in the Debugger window.
`Debug.RunTimeCheck.DataRangeChecks`	Boolean	RW	If set to `true` (default), check for data range violations during a debug session. Equivalent to the **Data Range** check box in the Debugger window.
`Debug.RunTimeCheck.StateInconsistencies`	Boolean	RW	If set to `true` (default), check for state inconsistencies during a debug session. Equivalent to the **State Inconsistency** check box in the Debugger window.
`Debug.RunTimeCheck.TransitionConflicts`	Boolean	RW	If set to `true` (default), check for transition conflicts during a debug session. Equivalent to the **Transition Conflict** check box in the Debugger window.
`Description`	String	RW	Description of this state (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this machine.
`Dirty`	Boolean	RO	If `true` (default), this model has changed since it was opened or saved.
`Document`	String	RW	Document link to this machine (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this machine.
`FullFileName`	String	RO	Full path name of file (default value = `''`) under which this machine (model) is stored.
`Iced`	Boolean	RO	Equivalent to property `Locked` (default = `false`) except that this property is used internally to lock this model from being changed during activities such as simulation.

Property	Type	Acc	Description
`Id`	Integer	RO	Unique identifier assigned to this machine to distinguish it from other objects loaded in memory.
`isLibrary`	Boolean	RO	If `true` (default = `false`), specifies that the current model builds a library and not an application.
`Locked`	Boolean	RW	If set to `true` (default = `false`), prevents user from changing any Stateflow chart in this model.
`Machine`	Machine	RO	A handle to the Machine object for this Machine object, that is, this Machine object.
`Modified`	String	RW	Comment area (default = `''`) for entering date and name of modification to this model.
`Name`	String	RO	Name of this model (default = `'untitled'`) set when saved to disk.
`SfVersion`	Double	RO	Full version number for current Stateflow. For example, the string `'41112101'` appears for Stateflow version 4.1.1 and MATLAB version 12.1. The remaining `'01'` is for internal use.
`Tag`	Any Type	RW	A field you can use to hold data of any type for this machine (default = `[]`).
`Version`	String	RW	Comment string (default = `'none'`) for recording the version of this model.

Machine Methods

Machine objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Machine Properties” on page 15-11.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this machine.
`dialog`	Display the properties dialog of this machine.
`disp`	Display the property names and their settings for this Machine object.
`find`	Find all objects that this machine contains that meet the specified criteria. **Note** Do not use the `-depth` switch with the `find` method for a machine object.
`findDeep`	Return all objects of the specified type in this machine at all levels of containment (i.e., infinite depth).
`findShallow`	Return all objects of the specified type in this machine at only the first level of containment (i.e., first depth).
`get`	Return the specified property settings for this machine.
`help`	Display a list of properties for this Machine object with short descriptions.
`methods`	Display all nonglobal methods of this Machine object.
`parse`	Parse all the charts in this machine.

Method	Description
`set`	Set the specified property of this Machine object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Machine object.

Chart Properties

Stateflow API objects of type Chart have the properties shown below. See also “Chart Methods” on page 15-22.

Property	Type	Acc	Description
`ChartColor`	[R,G,B]	RW	Background color of this chart in a 1-by-3 RGB array (default = `[1 0.9608 0.8824]`)with each value normalized on a scale of 0 to 1.
`Debug. Breakpoints. OnEntry`	Boolean	RW	If set to `true` (default = `false`), set the chart entry breakpoint for this chart. Equivalent to selecting the **Chart Entry** check box in the properties dialog for this chart.
`Decomposition`	Enum	RW	Set this property to `'EXCLUSIVE_OR'` (default) to specify exclusive (OR) decomposition for the states at the first level of containment in this chart. Set to `'PARALLEL_AND'` to specify parallel (AND) decomposition for these states. Equivalent to the **Decomposition** selection in the context menu for the Stateflow diagram editor.
`Description`	String	RW	Description (default = `''`) of this state. Equivalent to entering a description in the **Description** field of the properties dialog for this chart.
`Dirty`	Boolean	RW	If set to `true` (default = `false`), this chart has changed since being opened or saved.
`Document`	String	RW	Document link (default = `''`) to this chart. Equivalent to entering the **Document Link** field of the properties dialog for this chart.
`Editor`	Editor	RO	Editor object for this chart.

Property	Type	Acc	Description
`EnableBitOps`	Boolean	RW	If set to `true` (default = `false`), enables C-like bit operations in generated code for this chart. Equivalent to the **Enable C-like bit operations** check box in the chart properties dialog.
`ErrorColor`	[R,G,B]	RW	Set the RGB color for errors in the Stateflow Diagram Editor in a 1-by-3 RGB array (default value `[1 0 0]`) with each value normalized on a scale of 0 to 1. Equivalent to changing the **Error** color in the **Colors & Fonts** dialog under **Edit** > **Style**.
`ExecuteAt Initialization`	Boolean	RW	If set to `true` (default = `false`), this chart's state configuration is initialized at time zero instead of at the first input event. Equivalent to selecting the **Execute (enter) Chart at Initialization** check box in chart properties dialog
`ExportChart Functions`	Boolean	RW	If set to `true` (default = `false`), graphical functions at chart level are made global. Equivalent to selecting the **Export Chart Level Graphical Functions (Make Global)** check box in chart properties dialog.
`Iced`	Boolean	RO	Equivalent to property `Locked` (default = `false`) except that this property is used internally to lock this chart from change during activities such as simulation.
`Id`	Integer	RO	Unique identifier assigned to this chart to distinguish it from other objects loaded in memory.
`JunctionColor`	[R,G,B]	RW	Set the RGB color for junctions in the Stateflow Diagram Editor in a 1-by-3 RGB array (default value `[0.6824 0.3294 0]`) with each value normalized on a scale of 0 to 1. Equivalent to changing the **Junction** color in the **Colors & Fonts** dialog under **Edit** > **Style**.

Property	Type	Acc	Description
`Locked`	Boolean	RW	If set to `true` (default = `false`), mark this chart as read-only and prohibit any write operations on it. Equivalent to selecting the **Locked** check box in the **Editor** section of the properties dialog for this chart.
`Machine`	Machine	RO	Machine that contains this chart.
`Name`	String	RW	Name of this chart (default = `'Chart'`). Equivalent to changing the name of this chart's Stateflow block in Simulink.
`NoCodegenFor CustomTargets`	Boolean	RW	If set to `true` (default = `false`), no code is generated for this chart for custom targets (only for the simulation target, `sfun`). Equivalent to the **No Code Generation for Custom Targets** check box in the properties dialog for this chart.
`SampleTime`	String	RW	Sample time for activating this chart (default = `''`). Applies only when the `UpdateMethod` property for this chart is set to `'DISCRETE'` (= **Sampled** in the **Update method** field in the properties dialog for this chart).
`SelectionColor`	[R,G,B]	RW	Color of selected items for this chart in a 1-by-3 RGB array (default value `[1 0 0.5176]`) with each value normalized on a scale of 0 to 1. Equivalent to changing the **Selection** color in the **Colors & Fonts** dialog under **Edit** > **Style**.
`StateColor`	[R,G,B]	RW	Color of the state box in a 1-by-3 RGB array (default value `[0 0 0]`) with each value normalized on a scale of 0 to 1. Equivalent to changing the **State/Frame** color in the **Colors & Fonts** dialog under **Edit** > **Style**.

Property	Type	Acc	Description
`StateFont.Angle`	Enum	RW	Font angle for the labels of State, Box, Function, and Note objects. Can be `'ITALIC'` or `'NORMAL'` (default). Equivalent to **Italic** and **Regular** settings when changing the font style of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with property `StateFont.Weight` to achieve Bold Italic style. You can individually override this property with the `Font.Angle` property for Note objects.
`StateFont.Name`	String	RW	Font style (default = `'Helvetica'`) used for the labels of State, Box, Function, and Note objects. Enter a string for the font name (there are no selectable values). Font remains set to previous font for unrecognized font strings. Equivalent to changing the font of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.
`StateFont.Size`	Integer	RW	Font size (default = `12`) for the labels of State, Box, Function, and Note objects. Equivalent to changing the font size of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. You can individually override this property with the `FontSize` property for State, Box, and Function objects and with the `Font.Size` property for Note objects.
`StateFont.Weight`	Enum	RW	Font weight for state labels. Can be `'BOLD'` or `'NORMAL'` (default). Equivalent to the **Bold** and **Regular** settings of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with the property `StateFont.Angle` to achieve Bold Italic style. You can individually override this property with the `Font.Weight` property for Note objects.

Property	Type	Acc	Description
`StateLabelColor`	[R,G,B]	RW	Color of the state labels for this chart in a 1-by-3 RGB array (default = `[0 0 0]`) with each value normalized on a scale of 0 to 1. Equivalent to changing the label color of **StateLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.
`StrongDataTyping WithSimulink`	Boolean	RW	If set to `true` (default = `false`), set strong data typing with Simulink I/O. Equivalent to selecting the **Use Strong Data Typing with Simulink I/O** check box in the chart properties dialog.
`Tag`	Any Type	RW	A field you can use to hold data of any type for this chart (default = `[]`).
`TransitionColor`	[R,G,B]	RW	Set the RGB color for transitions in the Stateflow Diagram Editor in a 1-by-3 RGB array (default = `[0.2902 0.3294 0.6039]`) with each value normalized on a scale of 0 to 1. Equivalent to changing the **Transition** color in the **Colors & Fonts** dialog under **Edit** > **Style**.
`TransitionFont. Angle`	Enum	RW	Font angle for state labels. Can be `'ITALIC'` or `'NORMAL'` (default). Equivalent to **Italic** and **Regular** settings when you change the font style of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with property `StateFont.Weight` to achieve Bold Italic style.
`TransitionFont. Name`	String	RW	Font style (default = `'Helvetica'`) used for transition labels. Enter a string for font name (there are no selectable values). Font remains set to previous font for unrecognized font strings. Equivalent to changing the font of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.

Property	Type	Acc	Description
`TransitionFont.Size`	Integer	RW	Default font size (default = `12`) for transition labels. Truncated to closest whole number less than or equal to entered value. Equivalent to changing the font size of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.
`TransitionFont.Weight`	Enum	RW	Font weight for transition labels. Can be `'BOLD'` or `'NORMAL'` (default). Equivalent to **Bold** and **Regular** settings when you change the font style of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**. Use with property `StateFont.Angle` to achieve Bold Italic style.
`TransitionLabelColor`	[R,G,B]	RW	Color of the transition labels for this chart in a 1-by-3 RGB array (default = `[0.2902 0.3294 0.6039]`) with each value normalized on a scale of 0 to 1. Equivalent to changing the label color of **TransitionLabel** in the **Colors & Fonts** dialog under **Edit** > **Style**.
`UpdateMethod`	Enum	RW	Activation method of this chart. Can be `'INHERITED'` (default), `'DISCRETE'`, or `'CONTINUOUS'`. Equivalent to the **Update method** field in the properties dialog for this chart, which takes one of the following corresponding selections: **Triggered or Inherited**, **Sampled**, **Continuous**.
`Visible`	Boolean	RW	If set to `true` (default) , display this chart in the chart diagram editor.

Chart Methods

Chart objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Chart Properties” on page 15-16.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this chart.
`defaultTransitions`	Return the default transitions in this chart at the top level of containment.
`dialog`	Display the properties dialog of this chart.
`disp`	Display the property names and their settings for this Chart object.
`find`	Find all objects that this chart contains that meet the specified criteria.
`findDeep`	Return all objects of the specified type in this chart at all levels of containment (i.e., infinite depth).
`findShallow`	Return all objects of the specified type in this chart at only the first level of containment (i.e., first depth).
`get`	Return the specified property settings for this chart.
`help`	Display a list of properties for this Chart object with short descriptions.
`methods`	Display all nonglobal methods of this Chart object.
`parse`	Parse this chart.
`set`	Set the specified property of this Chart object with the specified value.

Method	Description
`struct`	Return and display a MATLAB structure containing the property settings of this Chart object.
`view`	Display this chart in a Stateflow diagram editor.

State Properties

Stateflow API objects of type State have the properties listed in the table below. See also "State Methods" on page 15-27.

Property	Type	Acc	Description
`ArrowSize`	Double	RW	Size of transition arrows coming into this state (default = 8). Equivalent to selecting **Arrowhead Size** from the context menu for this state.
`BadIntersection`	Boolean	RO	If `true`, this state graphically intersects a box, graphical function, or other state.
`Chart`	Chart	RO	Chart object containing this state.
`Debug. Breakpoints. onDuring`	Boolean	RW	If set to `true` (default = `false`), set the state entry breakpoint for this chart. Equivalent to selecting the **State During** check box in the properties dialog for this state.
`Debug. Breakpoints. OnEntry`	Boolean	RW	If set to `true` (default = `false`), set the state entry breakpoint for this chart. Equivalent to selecting the **State Entry** check box in the properties dialog for this state.
`Debug. Breakpoints. onExit`	Boolean	RW	If set to `true` (default = `false`), set the state entry breakpoint for this chart. Equivalent to selecting the **State Exit** check box in the properties dialog for this state.
`Decomposition`	Enum	RW	Set this property to `'EXCLUSIVE_OR'` (default) to specify exclusive (OR) decomposition for the states at the first level of containment in this state. Set to `'PARALLEL_AND'` to specify parallel (AND) decomposition for these states. Equivalent to the **Decomposition** selection in the context menu for the state.

Property	Type	Acc	Description
`Description`	String	RW	Description of this state (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this state.
`Document`	String	RW	Document link to this state (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this state.
`FontSize`	Double	RW	Size of the font (default = `12`) for the label text for this state. This property overrides the default size for this state, which is set by the `StateFont.Size` property of the Chart object containing this state. Equivalent to selecting **Font Size** > **<font size>** in the context menu for this state.
`Id`	Integer	RO	Unique identifier assigned to this state to distinguish it from other objects loaded in memory.
`IsGrouped`	Boolean	RW	If set to `true` (default = `false`), group this state. Nothing is allowed to change inside a grouped state. This property is also useful for copying states to a new location. See "Copying by Grouping (Recommended)" on page 13-30.
`IsSubchart`	Boolean	RW	If set to `true` (default = `false`), make this state a subchart.
`LabelString`	String	RW	Label for this state (default = `'?'`). Equivalent to typing the label for this state in its label text field in the diagram editor.
`Machine`	Machine	RO	Machine containing this state.
`Name`	String	RW	Name of this state (default = `''`). Equivalent to typing this state's name into the beginning of the label text field for this state in the diagram editor. `Name` is separated from the remainder of this state's label text by a forward slash (`/`) character.

Property	Type	Acc	Description
`Position`	Rect	RW	Position and size of this state's box in the Stateflow chart, given in the form of a 1-by-4 array (default is `[0 0 90 60]`) consisting of the following: • (x,y) coordinates for the box's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the box
`Subviewer`	Chart or State	RO	State or chart in which this state can be graphically viewed.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this state.
`Type`	Enum	RO	Type of this state (`default = 'OR'`). Can be `'OR'` (exclusive) or `'AND'` (parallel). The type of this state is determined by the parent's `Decomposition` property.

State Methods

State objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “State Properties” on page 15-24.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this state.
`defaultTransitions`	Return the default transitions in this state at the top level of containment.
`delete`	Delete this state.
`dialog`	Display the properties dialog of this state.
`disp`	Display the property names and their settings for this State object.
`find`	Find all objects that this state contains that meet the specified criteria.
`findDeep`	Return all objects of the specified type in this state at all levels of containment (i.e., infinite depth).
`findShallow`	Return all objects of the specified type in this state at only the first level of containment (i.e., first depth).
`get`	Return the specified property settings for this state.
`help`	Display a list of properties for this State object with short descriptions.
`innerTransitions`	Return the inner transitions that originate with this state and terminate on a contained object.
`methods`	Display all nonglobal methods of this State object.
`outerTransitions`	Return an array of transitions that exit the outer edge of this state and terminate on an object outside the containment of this state.

Method	Description
`outputData`	Output the activity status of this state to Simulink via a data output port on the chart block of this state.
`set`	Set the specified property of this State object with the specified value.
`sourcedTransitions`	Return all inner and outer transitions whose source is this state.
`struct`	Return and display a MATLAB structure containing the property settings of this State object.
`view`	Display this state's chart in a diagram editor with this state highlighted.

Box Properties

The following are properties of Stateflow API objects of type Box. See also “Box Methods” on page 15-31.

Property	Type	Acc	Description
`ArrowSize`	Double	RW	Size of transition arrows coming into this box (default = 8). Equivalent to selecting **Arrowhead Size** from the context menu for this box.
`BadIntersection`	Boolean	RO	If `true`, this box graphically intersects a state, graphical function, or other box.
`Chart`	Chart	RO	Chart object containing this box.
`Description`	String	RW	Description of this box (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this box.
`Document`	String	RW	Document link to this box (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this box.
`FontSize`	Double	RW	Size of the font (default = `12`) for the label text of this box . This property overrides the default size for this box, which is set by the `StateFont.Size` property of the Chart object containing this box. Equivalent to selecting **Font Size** > **<font size>** in the context menu for this box.
`Id`	Integer	RW	Unique identifier assigned to this box to distinguish it from other objects loaded in memory.
`IsGrouped`	Boolean	RW	If set to `true` (default = `false`), group this box.
`IsSubchart`	Boolean	RW	If set to `true` (default = `false`), make this box a subchart.

Property	Type	Acc	Description
`LabelString`	String	RW	Label for this box (default = `'?'`). Equivalent to typing the label for this box in its label text field in the diagram editor.
`Machine`	Machine	RO	Machine that contains this box.
`Name`	String	RW	Name of this box (default = `''`). Equivalent to typing this box's name into the beginning of the label text field for this box in the diagram editor.
`Position`	Rect	RW	Position and size of this box in the Stateflow chart, given in the form of a 1-by-4 array (default is `[0 0 90 60]`) consisting of the following: • (x,y) coordinates for the box's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the box
`Subviewer`	Chart or State	RO	State or chart in which this box can be graphically viewed.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this box.

Box Methods

Box objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Box Properties” on page 15-29.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this box.
`defaultTransitions`	Return the default transitions in this box at the top level of containment.
`delete`	Delete this box.
`dialog`	Display the properties dialog of this box.
`disp`	Display the property names and their settings for this Box object.
`find`	Find all objects that this box contains that meet the specified criteria.
`findDeep`	Return all objects of the specified type in this box at all levels of containment (i.e., infinite depth).
`findShallow`	Return all objects of the specified type in this box at only the first level of containment (i.e., first depth).
`get`	Return the specified property settings for this box.
`help`	Display a list of properties for this Box object with short descriptions.
`innerTransitions`	Return the inner transitions that originate with this box and terminate on a contained object.
`methods`	Display all nonglobal methods of this Box object.
`outerTransitions`	Return an array of transitions that exit the outer edge of this box and terminate on an object outside the containment of this box.

Method	Description
`set`	Set the specified property of this Box object with the specified value.
`sourcedTransitions`	Return all inner and outer transitions whose source is this box.
`struct`	Return and display a MATLAB structure containing the property settings of this Box object.
`view`	Display this box's chart in a diagram editor with this box highlighted.

Function Properties

Stateflow API objects of type Function have the properties listed in the table below. See also “Function Methods” on page 15-35.

Property	Type	Acc	Description
`ArrowSize`	Double	RW	Size of transition arrows coming into this function (default = `8`). Equivalent to selecting **Arrowhead Size** from the context menu for this function.
`BadIntersection`	Boolean	RO	If `true`, this state graphically intersects a state, box, or other graphical function.
`Chart`	Chart	RO	Chart object containing this function.
`Description`	String	RW	Description of this function (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this function.
`Document`	String	RW	Document link to this function. Equivalent to entering the **Document Link** field of the properties dialog for this function.
`FontSize`	Double	RW	Size of the (default = `12`) font of the label text for this function. This property overrides the default size for this function, which is set by the `StateFont.Size` property of the Chart object containing this function. Equivalent to selecting **Font Size** > **<font size>** in the context menu for this function.
`Id`	Integer	RO	Unique identifier assigned to this function to distinguish it from other objects in the model.
`IsGrouped`	Boolean	RW	If set to `true` (default = `false`), group this function.
`IsSubchart`	Boolean	RW	If set to `true` (default = `false`), make this function a subchart.

Property	Type	Acc	Description
`LabelString`	String	RW	Label for this function (default = `'()'`). Equivalent to typing the label for this function in its label text field in the diagram editor.
`Machine`	Machine	RO	Machine that contains this function.
`Name`	String	RW	Name of this function (default = `''`). Equivalent to typing this function's name into the beginning of the label text field after the word `'function'` in the diagram editor.
`Position`	Rect	RW	Position and size of this function's box in the Stateflow chart, given in the form of a 1-by-4 array (default is `[0 0 90 60]`) consisting of the following: • (x,y) coordinates for the box's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the box
`Subviewer`	Chart or State	RO	State or chart in which this function can be graphically viewed.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this function.

Function Methods

Function objects have the methods displayed in the table below. For details on each method, see the "API Methods Reference" on page 16-1.

See also see "Function Properties" on page 15-33.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this function.
`defaultTransitions`	Return the default transitions in this function at the top level of containment.
`delete`	Delete this function.
`dialog`	Display the properties dialog of this function.
`disp`	Display the property names and their settings for this Function object.
`find`	Find all objects that this graphical function contains that meet the specified criteria.
`findDeep`	Return all objects of the specified type in this function at all levels of containment (i.e., infinite depth).
`findShallow`	Return all objects of the specified type in this function at only the first level of containment (i.e., first depth).
`get`	Return the specified property settings for this function.
`help`	Display a list of properties for this Function object with short descriptions.
`innerTransitions`	Return the inner transitions that originate with this function and terminate on a contained object.
`methods`	Display all nonglobal methods of this Function object.

Method	Description
`outerTransitions`	Return an array of transitions that exit the outer edge of this function and terminate on an object outside the containment of this function.
`set`	Set the specified property of this Function object with the specified value.
`sourcedTransitions`	Return all inner and outer transitions whose source is this function.
`struct`	Return and display a MATLAB structure containing the property settings of this Function object.
`view`	Display this function's chart in a diagram editor with this state highlighted.

Truth Table Properties

Stateflow API objects of type Function have the properties listed in the table below. See also “Truth Table Methods” on page 15-40.

Property	Type	Acc	Description
`ActionTable`	Cell Array	RW	A cell array of strings containing the contents of the Action Table for this truth table.
`BadIntersection`	Boolean	RO	If `true`, this truth table graphically intersects a state, box, graphical function, or other truth table.
`Chart`	Chart	RO	Chart object containing this truth table.
`ConditionTable`	Cell Array	RW	A cell array of strings containing the contents of the Condition Table for this truth table, including the **Actions** row.
`Description`	String	RW	Description of this truth table (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this truth table.
`Document`	String	RW	Document link to this truth table. Equivalent to entering the **Document Link** field of the properties dialog for this truth table.
`FontSize`	Double	RW	Size of the (default = `12`) font of the label text for this truth table. This property overrides the default size for this truth table, which is set by the `StateFont.Size` property of the Chart object containing this truth table. Equivalent to selecting **Font Size** > *font size* in the context menu for this truth table.
`Id`	Integer	RO	Unique identifier assigned to this truth table to distinguish it from other objects in the model.

Property	Type	Acc	Description
`LabelString`	String	RW	Full label for this truth table(default = `'()'`) including its return, name, and arguments. Equivalent to typing the label for this truth table in its label text field in the diagram editor.
`Machine`	Machine	RO	Machine that contains this truth table.
`Name`	String	RW	Name of this truth table (default = `''`). Equivalent to typing a name for this truth table into the label text field of the truth table box in the diagram editor. Label syntax is *return* = Name (*arguments*).
`OverSpecDiagnostic`	String	RW	Interprets the error diagnosis of this truth table as overspecified according to the possible values 'Error', 'Warning', or 'None'. In the truth table editor, the value of this property is assigned by selecting **Overspecified** from the **Diagnostics** menu item and then selecting one of the three values.
`Position`	Rect	RW	Position and size of this truth table's box in the Stateflow chart, given in the form of a 1-by-4 array (default is `[0 0 90 60]`) consisting of the following: • (x,y) coordinates for the box's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the box
`Subviewer`	Chart or State	RO	State or chart in which this truth table can be graphically viewed.

Property	Type	Acc	Description
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this truth table.
`UnderSpecDiagnostic`	String	RW	Interprets the error diagnosis of this truth table as underspecified according to the possible values 'Error', 'Warning', or 'None'. In the truth table editor, the value of this property is assigned by selecting **Underspecified** from the **Diagnostics** menu item and then selecting one of the three values.

Truth Table Methods

Truth table objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also see “Truth Table Properties” on page 15-37.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this truth table.
`delete`	Delete this truth table.
`dialog`	Display the properties dialog of this truth table.
`disp`	Display the property names and their settings for this truth table object.
`find`	Find all objects that this graphical truth table contains that meet the specified criteria.
`findDeep`	Return all objects of the specified type in this truth table at all levels of containment (i.e., infinite depth).
`findShallow`	Return all objects of the specified type in this truth table at only the first level of containment (i.e., first depth).
`get`	Return the specified property settings for this truth table.
`help`	Display a list of properties for this truth table object with short descriptions.
`methods`	Display all nonglobal methods of this truth table object.
`set`	Set the specified property of this truth table object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this truth table object.
`view`	Display this truth table’s chart in a diagram editor with this state highlighted.

Note Properties

Stateflow API objects of type Note have the properties listed in the table below. See also “Note Methods” on page 15-43.

Property	Type	Acc	Description
`Alignment`	Enum	RW	Alignment of text in note box. Can be `'LEFT'` (default), `'CENTER'`, or `'RIGHT'`.
`Chart`	Chart	RO	Chart object containing this note.
`Description`	String	RW	Description of this note (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this note.
`Document`	String	RW	Document link to this note (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this note.
`Font.Name`	String	RO	Name of the font (default = `'Helvetica'`) for the text in this note. This property is read-only (RO) and set by the `StateFont.Name` property of the Chart object containing this note.
`Font.Angle`	String	RW	Style of the font for the text in this note. Can be `'ITALIC'` or `'NORMAL'` (default). This property overrides the default style for this note, which is set by the `StateFont.Angle` property of the Chart object containing this note.
`Font.Size`	Double	RW	Size of the font (default = `12`) for the label text for this note. This property overrides the default size for this note, which is set by the `StateFont.Size` property of the Chart object containing this note. Equivalent to selecting **Font Size** > **<font size>** in the context menu for this note.

Property	Type	Acc	Description
`Font.Weight`	String	RW	Weight of the font for the label text for this note. Can be `'BOLD'` or `'NORMAL'` (default). This property overrides the default weight for the text in this note, which is set by the `StateFont.Weight` property of the Chart object containing this note.
`Id`	Integer	RO	Unique identifier assigned to this note to distinguish it from other objects in the model.
`Interpretation`	Enum	RW	How the text in this note is interpreted for text processing. Can be `'NORMAL'` (default) or `'TEX'`.
`Machine`	Machine	RO	Machine that contains this note.
`Position`	Rect	RW	Position and size of this note's box in the Stateflow chart, given in the form of a 1-by-4 array (default is `[0 0 25 25]`) consisting of the following: • (x,y) coordinates for the box's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the box
`Subviewer`	Chart or State	RO	State or chart in which this note can be graphically viewed.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this note.
`Text`	String	RW	Label for this note (default = `'?'`). The text content for this note that you enter directly into the note in the diagram editor or in the **Label** field of the properties dialog for this note.

Note Methods

Note objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Note Properties” on page 15-41.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this note.
`delete`	Delete this note.
`dialog`	Display the properties dialog of this note.
`disp`	Display the property names and their settings for this Note object.
`get`	Return the specified property settings for this note.
`help`	Display a list of properties for this Note object with short descriptions.
`methods`	Display all nonglobal methods of this Note object.
`set`	Set the specified property of this Note object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Note object.
`view`	Display this note’s chart in a diagram editor with this note highlighted.

Transition Properties

Stateflow API objects of type Transition have the properties listed in the table below. See also “Transition Methods” on page 15-47.

Property	Type	Acc	Description
`ArrowSize`	Double	RW	Size of the arrow (default = `10`) for this transition.
`Chart`	Chart	RO	Stateflow chart object containing this transition.
`Debug.Breakpoints.WhenTested`	Boolean	RW	If set to `true` (default = `false`) , set a debugging breakpoint to occur when this transition is tested to see whether it is a valid transition or not. Equivalent to selecting the **When Tested** check box in the properties dialog of this transition.
`Debug.Breakpoints.WhenValid`	Boolean	RW	If set to `true` (default = `false`) , set a debugging breakpoint to occur when this transition has tested as valid. Equivalent to selecting the **When Valid** check box in the properties dialog of this transition.
`Description`	String	RW	Description of this transition (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this transition.
`Destination`	State or Junction	RW	Destination state or junction (default = `[]`) of this transition. Assign `Destination` the destination object for this transition. You can also use the property `Destination` to detach the destination end of a transition, through the command `t.Destination = []` where `t` is the Transition object.
`DestinationOClock`	Double	RW	Location of transition destination connection on state (default = `0`). Varies from 0 to 12 for full clock cycle location. Its value is taken as modulus 12 of its assigned value.

Property	Type	Acc	Description
`Document`	String	RW	Document link to this transition (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this transition.
`DrawStyle`	Enum	RW	Set the drawing style for this transition. Set to `'STATIC'` (default) for static transitions or `'SMART'` for smart transitions. Equivalent to selecting the **Smart** switch toggle from the context menu for this transition.
`FontSize`	Double	RW	Size of the font (default = `12`) for the label text for this box. This property overrides the default size for this box, which is set by the `TransitionFont.Size` property of the Chart object containing this box. Equivalent to selecting **Font Size** > **<font size>** in the context menu for this box.
`Id`	Integer	RO	Unique identifier assigned to this transition to distinguish it from other objects loaded in memory.
`LabelPosition`	Rect	RW	Position and size of this note's box in the Stateflow chart, given in the form of a 1-by-4 array (default is `[0 0 8 14]`) consisting of the following: • (x,y) coordinates for the box's left upper vertex relative to the upper left vertex of the Stateflow diagram editor workspace • Width and height of the box
`LabelString`	String	RW	Label for this transition (default = `'?'`). Equivalent to typing the label for this transition in its label text field in the diagram editor.
`Machine`	Machine	RO	Machine containing this transition.
`MidPoint`	Rect	RW	Position of the midpoint of this transition relative to the upper left corner of the Stateflow diagram editor workspace in an `[x y]` point array (default = `[0 0]`).

Property	Type	Acc	Description
`Source`	State or Junction	RW	Source state or junction of this transition (default = `[]`). Assign `Source` the source object for this transition. You can also use the property `Source` to detach the source end of a transition, through the command `t.Source = []` where `t` is the Transition object.
`SourceEndPoint`	Rect	RO*	`[x y]` spatial coordinates for the endpoint of a transition (default = `[2 2]`). This property is RW (read/write) only for default transitions. For all other transitions it is RO (read-only).
`SourceOClock`	Double	RW	Location of transition source connection on state (default = `0`). Varies from 0 to 12 for full clock cycle location. The value taken for this property is the modulus 12 of the entered value.
`Subviewer`	Chart or State	RO	State or chart in which this transition can be graphically viewed.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this transition.

Transition Methods

Transition objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Transition Properties” on page 15-44.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this transition.
`delete`	Delete this transition.
`dialog`	Display the properties dialog of this transition.
`disp`	Display the property names and their settings for this Transition object.
`get`	Return the specified property settings for this transition.
`help`	Display a list of properties for this Transition object with short descriptions.
`methods`	Display all nonglobal methods of this Transition object.
`set`	Set the specified property of this Transition object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Transition object.
`view`	Display this transition’s chart in a diagram editor with this transition highlighted.

Junction Properties

Stateflow API objects of type Junction have the properties listed in the table below. See also “Junction Methods” on page 15-49.

Property	Type	Acc	Description
`ArrowSize`	Double	RW	Size of transition arrows (default = `8`) coming into this junction.
`Chart`	Chart	RO	Chart that this junction resides in.
`Description`	String	RW	Description of this junction (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this junction.
`Document`	String	RW	Document link to this junction (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this junction.
`Id`	Integer	RO	Unique identifier assigned to this junction to distinguish it from other objects loaded in memory.
`Machine`	Machine	RO	Machine containing this junction.
`Position. Center`	Rect	RW	Position of the center of this junction (default = `[10 10]`) relative to the upper left corner of the parent chart or state as an `[x,y]` point array.
`Position. Radius`	Rect	RO	Radius of this junction (default = `10`).
`Subviewer`	Chart or State	RO	State or chart in which this junction can be graphically viewed.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this junction.
`Type`	Enum	RO	Type of this junction. For junctions, can be `'CONNECTIVE'`(default) or `'HISTORY'`

Junction Methods

Junction objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Junction Properties” on page 15-48.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this junction.
`delete`	Delete this junction.
`dialog`	Display the properties dialog for this junction
`disp`	Display the property names and their settings for this Junction object.
`get`	Return the specified property settings for this junction.
`help`	Display a list of properties for this Junction object with short descriptions.
`methods`	Display all nonglobal methods of this Junction object.
`set`	Set the specified property of this Junction object with the specified value.
`sourcedTransitions`	Return all inner and outer transitions whose source is this junction.
`struct`	Return and display a MATLAB structure containing the property settings of this Junction object.
`view`	Display this junction’s chart in a diagram editor with this junction highlighted.

Data Properties

Stateflow API objects of type Data have the properties listed in the table below. See also "Data Methods" on page 15-55.

Property	Type	Acc	Description
`DataType`	Enum	RW	Data type of this data. Can be `'double'` (default), `'single'`, `'int32'`, `'int16'`, `'int8'`, `'uint32'`, `'uint16'`, `'uint8'`, `'boolean'`, `'fixpt'`, or `'ml'`. Equivalent to an entry in the **Type** column for this data in Explorer or the **Type** field in the properties dialog for this data.
`Debug.Watch`	Boolean	RW	If set to `true` (default = `false`), causes the debugger to halt execution if this data is modified. Setting this property to `true` is equivalent to selecting the **Watch** column entry for this data in the Explorer or selecting the **Watch in debug** check box in the properties dialog for this data.
`Description`	String	RW	Description of this data (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this data.
`Document`	String	RW	Document link to this data (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this data.
`Id`	Integer	RO	Unique identifier assigned to this data to distinguish it from other objects loaded in memory.
`FixptType.Bias`	Double	RW	The Bias value for this fixed-point type (default = `0`).
`FixptType.FractionalSlope`	Double	RW	The Fractional Slope value for this fixed-point type (default = `1`).
`FixptType.RadixPoint`	Integer	RW	The power of 2 specifying the binary-point location for this fixed-point type (default = 0).

Property	Type	Acc	Description
`FixptType.BaseType`	Enum	RW	The size and sign of the base integer type for the quantized integer, Q, representing this fixed-point type. Can be `'int32'`, `'int16'`, `'int8'`, `'uint32'`, `'uint16'`, or `'uint8'` (default).
`InitFromWorkspace`	Boolean	RW	If set to `true` (default = `false`), this data is initialized from the MATLAB workspace. Setting this property to `true` is equivalent to selecting the **FrWS** column entry for this data in the Explorer or setting the **Initialize from** field to **workspace** in the properties dialog for this data.
`Machine`	Machine	RO	Machine that contains this data.
`Name`	String	RW	Name of this data (default = `'data`*n*`'`, where *n* is a counter of data with the name root `data`). Equivalent to entering the name of this data in the **Name** field of its properties dialog. Also can be named (renamed) in the Explorer by double-clicking the entry in the **Name** column for this data and editing.
`OutputState`	State	RO	State whose activity this data represents as an output. Create the data for this state through the State method `outputData`. Equivalent to selecting **Output State Activity** property for this state.
`ParsedInfo.Array.Size`	Integer	RO	Numeric equivalent of Data property `Props.Array.Size`, a String (default = `[]`).
`ParsedInfo.Array.FirstIndex`	Integer	RO	Numeric equivalent of Data property `Props.Range.FirstIndex`, a String (default = `0`).
`ParsedInfo.InitialValue`	Double	RO	Numeric equivalent of Data property `Props.InitialValue`, a String (default = `0`).

Property	Type	Acc	Description
`ParsedInfo.Range.Maximum`	Double	RO	Numeric equivalent of Data property `Props.Range.Maximum`, a String (default = `inf`).
`ParsedInfo.Range.Minimum`	Double	RO	Numeric equivalent of Data property `Props.Range.Minimum`, a String (default = `-inf`).
`PortNumber`	Integer	RO	Port index number for this data (default = `1`).
`Props.Range.Maximum`	String	RW	Maximum value (default = `''`) that this data can have during execution or simulation of the state machine. Equivalent to entering value in the **Max** column for this data in Explorer or the **Max** field in the properties dialog for this data.
`Props.Range.Minimum`	String	RW	Minimum value (default = `''`) that this data can have during execution or simulation of the state machine. Equivalent to entering a value in the **Min** column for this data in Explorer or the **Min** field in the properties dialog for this data.
`Props.InitialValue`	String	RW	If the source of the initial value for this data is the Stateflow data dictionary, this is the value used (default = `0`) . Equivalent to entering this value in the **InitVal** column for this data in the Explorer or similar field in the properties dialog for this data.
`Props.Array.Size`	String	RW	Specifying a positive value for this property specifies that this data is an array of this size (default = `0`). Equivalent to entering a positive value in the **Size** column for this data in the Explorer or in the **Sizes** field of the properties dialog for this data.
`Props.Array.FirstIndex`	String	RW	Index of the first element of this data (default = `0`) if it is an array (that is, `Props.Array.Size` > `1`) . Equivalent to entering a value of zero or greater in the **First Index** field of the properties dialog for this data.

Property	Type	Acc	Description
`SaveToWorkspace`	Boolean	RW	If set to `true` (default = `false`), this data is saved to the MATLAB workspace. Setting this property to `true` is equivalent to selecting the **ToWS** column entry for this data in the Explorer or selecting the **Save final value to base workspace** field in the properties dialog for this data.
`Scope`	Enum	RW	Scope of this data. Allowed values vary with the object containing this data. The following apply to any data object: • `'LOCAL_DATA'` (Local, default) • `'CONSTANT_DATA'` (Constant) • `'OUTPUT_DATA'` (Output to Simulink) The following apply to data for machines only: • `'IMPORTED_DATA'` (Exported) • `'EXPORTED_DATA'` (Imported) The following apply to data for charts only: • `'INPUT_DATA'` (Input to Simulink) The following apply to data for charts and functions only: • `'TEMPORARY_DATA'` (Temporary) The following apply to data for functions only: • `'FUNCTION_INPUT_DATA'` (Input Data) • `'FUNCTION_OUTPUT_DATA'` (Output Data) Above values are equivalent to entering the value shown in parentheses in the **Scope** field of the properties dialog or the Explorer for this data.

Property	Type	Acc	Description
`Tag`	Any Type	RW	Holds data of any type for this Data object (default = `[]`).
`Units`	String	RW	Physical units corresponding to the value of this data object (default = `''`).

Data Methods

Data objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Data Properties” on page 15-50.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this data.
`delete`	Delete this data.
`dialog`	Display the properties dialog of this Data object.
`disp`	Display the property names and their settings for this Data object.
`get`	Return the specified property settings for this data.
`help`	Display a list of properties for this Data object with short descriptions.
`methods`	Display all nonglobal methods of this Data object.
`set`	Set the specified property of this Data object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Data object.
`view`	Display this data in the Data properties dialog.

Event Properties

Stateflow API objects of type Event have the properties listed in the table below. See also “Event Methods” on page 15-58.

Property	Type	Acc	Description
`Debug.Breakpoints.StartBroadcast`	Boolean	RW	If set to `true` (default = `false`), set a debugger breakpoint for the start of the broadcast of this event. Equivalent to selecting the **Start of broadcast** check box in the properties dialog for this event.
`Debug.Breakpoints.EndBroadcast`	Boolean	RW	If set to `true` (default = `false`), set a debugger breakpoint for the end of the broadcast of this event. Equivalent to selecting the **End of broadcast** check box in the properties dialog for this event.
`Description`	String	RW	Description of this event (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this event.
`Document`	String	RW	Document link to this event (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this event.
`Id`	Integer	RO	Unique identifier assigned to this event to distinguish it from other objects loaded in memory.
`Machine`	Machine	RO	Machine this event belongs to.
`Name`	String	RW	Name of this event (default = `event`*n*, where *n* is a counter of events with the name root `event`).). Equivalent to entering the name in the **Name** field of the properties dialog for this event.
`PortNumber`	Integer	RO	Port index number for this event (default = `1`).

Property	Type	Acc	Description
`Scope`	Enum	RW	Scope of this data. Allowed values vary with the object containing this data. The following apply to any event: • `'LOCAL_EVENT'` (Local - default) The following apply to events for charts only: • `'INPUT_EVENT'` (Input from Simulink) • `'OUTPUT_EVENT'` (Output to Simulink) The following apply to events for machines only: • `'IMPORTED_EVENT'` (Imported) • `'EXPORTED_EVENT'` (Exported) Above values are equivalent to entering the value shown in parentheses in the **Scope** field of the properties dialog or the Explorer for this data.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this event.
`Trigger`	Enum	RW	Type of signal that triggers this **Input to Simulink** or **Output to Simulink** event associated with its chart. Can be one of the following: • `'EITHER_EDGE_EVENT'` (Either Edge - default) • `'RISING_EDGE_EVENT'` (Rising Edge) • `'FALLING_EDGE_EVENT'` (Falling Edge) • `'FUNCTION_CALL_EVENT'` (Function Call) Equivalent to specifying the indicated parenthetical expression for the **Trigger** field of the properties dialog for this event.

Event Methods

Event objects have the methods displayed in the table below. For details on each method, see the "API Methods Reference" on page 16-1.

See also "Event Properties" on page 15-56.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this event.
`delete`	Delete this event.
`dialog`	Display the properties dialog for this event.
`disp`	Display the property names and their settings for this Event object.
`get`	Return the specified property settings for this event.
`help`	Display a list of properties for this Event object with short descriptions.
`methods`	Display all nonglobal methods of this Event object.
`set`	Set the specified property of this Event object with the specified value.
`struct`	Return and display a MATLAB structure containing the property settings of this Event object.
`view`	Display this event in its properties dialog.

Target Properties

Stateflow API objects of type Target have the properties listed in the table below. See also “Target Methods” on page 15-63.

Property	Type	Acc	Description
`ApplyToAllLibs`	Boolean	RW	If set to `true` (default), use settings in this target for all libraries. Equivalent to selecting the **Use settings for all libraries** check box in this target's Target Builder dialog.
`CodeFlagsInfo`	Array	RO	A MATLAB vector of structures containing information on the code flag settings for this target. See special topic “CodeFlagsInfo Property of Targets” on page 15-60 for more information.
`CodegenDirectory`	String	RW	Directory to receive generated code (default = `''`). Applies only to targets other than `sfun` and `rtw` targets.
`CustomCode`	String	RW	Custom code included at the top of the generated code (default = `''`). Equivalent to the setting of the **Custom code included at the top of generated code** selection of the Target Options dialog for this target.
`CustomInitializer`	String	RW	Custom initialization code (default = `''`). Equivalent to the setting of the **Custom initialization code (called from mdlInitialize)** selection of the Target Options dialog for this target. Applies only to `sfun` and `rtw` targets.
`CustomTerminator`	String	RW	Custom termination code (default = `''`). Equivalent to the setting of the **Custom termination code (called from mdlTerminate)** selection of the Target Options dialog for this target. Applies only to `sfun` and `rtw` targets.

Property	Type	Acc	Description
`Description`	String	RW	Description of this target (default = `''`). Equivalent to entering a description in the **Description** field of the properties dialog for this target.
`Document`	String	RW	Document link to this target (default = `''`). Equivalent to entering the **Document Link** field of the properties dialog for this target.
`Id`	Integer	RO	Unique identifier assigned to this Target object to distinguish it from other objects loaded in memory.
`Machine`	Machine	RO	Stateflow machine containing this target.
`Name`	String	RW	Name of this target (default = `'untitled'`). Equivalent to naming or renaming this target in the Explorer.
`Tag`	Any Type	RW	Holds data of any type (default = `[]`) for this target.
`UserIncludeDirs`	String	RW	Custom include directory paths (default = `''`). Equivalent to the setting of the **Custom include directory paths** selection of the Target Options dialog for this target.
`UserLibraries`	String	RW	Custom libraries (default = `''`). Equivalent to the setting of the **Custom libraries** selection of the Target Options dialog for this target.
`UserSources`	String	RW	Custom source files (default = `''`). Equivalent to the setting of the **Custom source files** selection of the Target Options dialog for this target.

CodeFlagsInfo Property of Targets

The `CodeFlagsInfo` property of a Target object is a read-only MATLAB vector of structures containing information on the code flag settings for its target.

Each element in the vector has the following MATLAB structure of information about a particular code flag:

Element	Type	Description
`name`	String	Short name for this flag
`type`	String	The type of the code flag
`description`	String	A description of this code flag
`defaultValue`	Boolean	The default value of this code flag upon creation of its target
`visible`	Boolean	Whether or not this flag is visible
`enable`	Boolean	Whether or not to enable this flag
`value`	Boolean	The value of the flag

The first element of each structure is a shorthand name for the individual flag that you set in the **Coder Options** dialog. For example, the name 'comments' actually refers to the dialog setting **Comments in generated code**. While the `CodeFlagsInfo` property is informational only, you can use these shorthand flag names in the methods `getCodeFlag` and `setCodeFlag` to access and change the values of a flag.

The names of each of the possible code flags in the `CodeFlagsInfo` property along with the name of the flag as it appears in the **Coder Options** dialog for the target are as follows:

Name in CodeFlagsInfo	Name in Properties Dialog	Target	Default Value
`debug`	**Enable debugging/animation**	`sfun`	Enabled
`overflow`	**Enable overflow detection (with debugging)**	`sfun`	Enabled
`echo`	**Echo expressions without semicolons**	`sfun`	Enabled

Name in CodeFlagsInfo	Name in Properties Dialog	Target	Default Value
`comments`	**Comments in generated code**	`rtw`, `custom`	Disabled Enabled
`preservenames`	**Preserve symbol names**	`rtw`, `custom`	Disabled
`preservenames withparent`	**Append symbol names with no mangling**	`rtw`, `custom`	Disabled
`exportcharts`	**Use chart names with no mangling**	`rtw`, `custom`	Disabled
`statebitsets`	**Use bitsets for storing state configuration**	`rtw`, `custom`	Disabled
`databitsets`	**Use bitsets for storing boolean data**	`rtw`, `custom`	Disabled
`ioformat`	Enumerated value can be one of the following: • 0 = **Use global input/output data** • 1 = **Pack input/output data into structures** • 2 = **Separate argument for input/output data**	`custom`	0
`initializer`	**Generate chart initializer function**	`custom`	Disabled
`multi instanced`	**Multi-instance capable code**	`custom`	Disabled
`ppcomments`	**Comments for Post-processing**	`custom`	Disabled

For detailed descriptions of each of the preceding code flags, see “Specifying Code Generation Options” on page 11-11.

Target Methods

Target objects have the methods displayed in the table below. For details on each method, see the “API Methods Reference” on page 16-1.

See also “Target Properties” on page 15-59.

Method	Description
`classhandle`	Return a read-only handle to the class (object type) used to define this target.
`build`	Build this target only for those portions of the target's charts that have changed since the last build (i.e., incrementally).
`delete`	Delete this target.
`dialog`	Display the properties dialog for this target.
`disp`	Display the property names and their settings for this Target object.
`get`	Return the specified property settings for this target.
`getCodeFlag`	Return the value of the specified code flag for this target.
`help`	Display a list of properties for this Target object with short descriptions.
`make`	Compile this target for only those portions of this target's charts that have changed since the last compile (i.e., incrementally). For a simulation target (`sfun`), a dynamic link library (`sfun.dll`) is compiled from the generated code.
`methods`	Display all nonglobal methods of this Target object.
`rebuildAll`	Completely rebuild this target.
`regenerateAll`	Completely regenerate code for this target.
`set`	Set the specified property of this Target object with the specified value.
`setCodeFlag`	Set the specified code flag for this target with the specified value.

Method	Description
`struct`	Return and display a MATLAB structure containing the property settings of this Target object.
`view`	Display this target in the Target properties dialog.

16

API Methods Reference

Description of Method Reference Fields (p. 16-2)	Describes the individual fields of description in the method reference pages that follow.
Methods — Alphabetical List (p. 16-3)	A comprehensive list of the Stateflow API methods and their purpose in alphabetical order.

Description of Method Reference Fields

This chapter contains references for each individual method in the Stateflow API. Each method has its own reference pages, which contain the following information:

- **Objects** — A list of objects that share this method. Many methods are available for different object types but usually share similar functionality.
- **Purpose** — A short phrase indicating the method's use.
- **Description** — A longer description of each method's use along with special considerations.
- **Syntax** — A representation of the call to the method using dot notation. Standard function notation is also allowed. See the section "Accessing the Properties and Methods of Objects" on page 13-17.
- **Arguments** — A list of arguments required or optionally available for the method. Optional arguments are surrounded by brackets: `[]`.
- **Returns** — The return type of the method. Methods can return values of all types, including other Stateflow objects.
- **Example** — A pertinent example emphasizing the method's functionality and its uniqueness among other similarly named methods.

Methods — Alphabetical List

The following table lists and describes the methods of the Stateflow API:

Method	Purpose
`build`	Build this target incrementally
`classhandle`	Provide a handle to the schema class of this object's type
`copy`	Copy the specified array of objects to the clipboard
`defaultTransitions`	Return the default transitions in this object at the top level of containment
`delete`	Delete this object
`dialog`	Open the Properties dialog of this object
`disp`	Display the properties and settings for this object
`find`	Return specified objects in this object at all levels of containment
`findDeep`	Return specified objects in this object at all levels of containment below this object
`findShallow`	Return specified objects in this object at the first level of containment below this object
`generate`	Generate code incrementally for this target
`get`	Return a MATLAB structure containing the property settings of this object
`getCodeFlag`	Return the specified code flag
`help`	Display the list of properties for this object along with short descriptions

Method	Purpose
`innerTransitions`	Return the inner transitions that originate with this chart or state and terminate on a contained object
`make`	Make (compile, link, load) this target incrementally with no code generation
`methods`	List the names of the methods belonging to this object
`outerTransitions`	Return an array of outer transitions for this state
`outputData`	Create, retrieve, or delete a data output to Simulink of this state's activity status
`parse`	Parse this chart
`pasteTo`	Paste the objects in the clipboard to the specified container object
`rebuildAll`	Completely rebuild this target
`regenerateAll`	Completely regenerate code for this target
`set`	Set specified properties with the specified values
`setCodeFlag`	Set the specified code flag to the value you specify
`sfexit`	Close all Simulink models containing Stateflow diagrams and exit the Stateflow environment
`sfhelp`	Display Stateflow online help
`sfnew`	Create a Simulink model containing an empty Stateflow block

Method	Purpose
`sfprint`	Display the visible portion of a Stateflow diagram
`sfsave`	Save the current state machine (Simulink model)
`sfversion`	Return the current version of Stateflow
`sourcedTransitions`	Return the transitions that have this object as their source
`stateflow`	Open the Stateflow model window
`Stateflow.Box`	Create a box for a parent chart, state, box, or function.
`Stateflow.Data`	Create a data for a parent machine, chart, state, box, or function.
`Stateflow.Event`	Create an event for a parent machine, chart, state, box, or function.
`Stateflow.Function`	Create a graphical function for a parent chart, state, box, or function.
`Stateflow.Junction`	Create a junction for a parent chart, state, box, or function.
`Stateflow.Note`	Create a note for a parent chart, state, box, or function.
`Stateflow.State`	Create a state for a parent chart, state, box, or function.
`Stateflow.Target`	Create a target for a parent machine.
`struct`	Return a MATLAB structure containing the property settings of this object
`view`	Make this object visible for editing
`zoomIn and zoomOut`	Zoom in or out on this chart

build

Purpose

Build this target incrementally

Syntax

```
thisTarget.build
```

Description

The `build` method incrementally builds this target. It performs the following activities:

- Parses all charts completely.
- Generates code for charts incrementally.
- For a simulation target (`sfun`), a dynamic link library (`sfun.dll`) is compiled from the generated code.

If a complete build has already taken place, the `build` method performs an incremental build that builds only those portions of the target corresponding to charts that have changed since the last build.

Arguments

Name	Description
`thisTarget`	The Target object to build

Returns

None

Example

If `t` is a Target object, the command `t.build` builds the target for the Stateflow charts that have changed in the target's model since the last build and/or code generation.

See Also

The methods `rebuildAll`, `generate`, `regenerateAll`, and `make`

Purpose

Provide a handle to the schema class of this object's type

Syntax

```
handle = thisObject.classhandle
```

Description

The `classhandle` method returns a read-only handle to the schema class of this object's type. You can use the `classhandle` method to provide information about the structure of each object type.

Arguments

`thisObject`	The object for which to return a handle. Can be any Stateflow object.

Returns

`handle`	Handle to schema class of this object.

Example

If `j` is a Junction object, the class handle of a Junction object is `j.classhandle`. You can see the class schema for a Junction object by using the following `get` command:

```
j.classhandle.get
```

Two member arrays of the displayed class schema are `Properties` and `Methods`. These two members are members of the schema class for every object.

List the class schema for Properties with the following command:

```
j.classhandle.Properties.get
```

Two displayed members of the Properties schema are `Name` and `DataType`. Finally, using the class handle for a junction, you can display the properties of a Junction object along with their data types with the following command:

```
get(j.classhandle.Properties,{'Name','DataType'})
```

copy

Purpose

Copy the specified array of objects to the clipboard

Syntax

```
cbObj.copy(objArray)
```

Description

The `copy` method copies the specified objects to the clipboard. Objects to copy are specified through a single argument array of objects.

Later, complete the copy operation by invoking the `pasteTo` method.

Arguments

`cbObj` The Clipboard object to copy to.

`objArray` Array of Stateflow objects to copy. These objects must conform to the following:

- The objects copied must be all graphical (states, boxes, functions, transitions, junctions) or all nongraphical (data, events, targets).
- If all objects are graphical, they must all be seen in the same subviewer.

Returns

None

Example

See “Copying Objects” on page 13-29.

Purpose

Return the default transitions in this object at the top level of containment

Syntax

```
defaultTransitions = thisObject.defaultTransitions
```

Description

The `defaultTransitions` method returns the default transitions in this object at the top level of containment.

Arguments

`thisObject`	The object for which to return default transitions. Can be an object of type Chart, State, Box, or Function.

Returns

`defaultTransitions`	Array of default transitions in this object at the top level of containment

Example

If state `A` contains state `A1`, and state `A1` contains state `A11`, and states `A1` and `A11` have default transitions attached to them, the `defaultTransitions` method of state `A` returns the default transition attached to state `A1`.

delete

Purpose

Delete this object

Syntax

```
thisObject.delete
```

Description

The `delete` method deletes this object from the model. This is true for all but objects of type Root, Chart, Clipboard, and Editor.

Arguments

`thisObject`	The object to delete. Can be an object of type Machine, State, Box, Function, Truth Table, Note, Transition, Junction, Data, Event, or Target.

Returns

None

Example

If a state `A` is represented by the State object `sA`, the command `sA.delete` deletes state `A`.

Purpose

Open the Properties dialog of this object

Syntax

```
thisObject.dialog
```

Description

The `dialog` method opens the Properties dialog of its object.

Arguments

`thisObject`	The object for which to open the properties dialog. Can be an object of type Machine, State, Box, Function, Truth Table, Note, Transition, Junction, Data, Event, or Target.

Returns

None

Example

If state `A` is represented by State object `sA`, the MATLAB statement `sA.dialog` opens the Properties dialog for state `A`.

disp

Purpose

Display the properties and settings for this object

Syntax

```
thisObject.disp
```

Description

The `disp` method displays the properties and settings for this object. This is true for all but objects of type Root and Clipboard.

Arguments

`thisObject`	The object to display properties and settings for. Can be an object of type Machine, Chart, State, Box, Function, Truth Table, Note, Transition, Junction, Data, Event, or Target.

Returns

None

Example

If a state `A` is represented by the State object `sA`, the command `sA.disp` displays the property names and their settings for state `A`.

Purpose

Return specified objects in this object at all levels of containment

Syntax

```
objArray = thisObject.find(Specifier,Value, ...)
```

Note You can also nest specifications using braces ({}).

Description

Using combinations of specifier-value argument pairs, the `find` method returns objects in this object that match the specified criteria. The specifier-value pairs can be property based or based on other attributes of the object such as its depth of containment. Specifiers can also be logical operators (`-and`, `-or`, etc.) that combine other specifier-value pairs.

By default, the `find` command finds objects at all depths of containment within an object. You can specify the maximum depth of search with the `-depth` specifier. However, the zeroth level of containment, i.e., the searched object itself, is always included if it happens to satisfy the search criteria.

If no arguments are specified, the `find` command returns all objects of this object at all levels of containment.

Arguments

`thisObject` The object for which to find contained objects. Can be an object of type Root, Machine, State, Box, Function, or Truth Table.

`'-and'` No value is paired to this specifier. Instead, this specifier relates a previous specifier-value pair to a following specifier-value pair in an AND relation.

`'-class'` Following value is a string class name of the class to search for. Use this option to find all objects whose class exactly matches a given class. To allow matches for subclasses of a given class, use the `-isa` specifier. Classes are specified as the string name (e.g., `'Stateflow.State'`, `'Stateflow.Transition'`, etc.) or as a handle to the class (see the method `classhandle`).

`'-depth'` Following value is an integer depth to search, which can be 0,1,2,...,infinite. The default search depth is infinite.

Note Do not use the `'-depth'` switch with the `find` method for a machine object.

`'-function'` Following value is a handle to a function that evaluates each object visited in the search. The function must always return a logical scalar value that indicates whether or not the value is a match. If no property is specified, the function is passed the handle of the current object in the search. If a property is specified, the function is passed the value of that property.

In the following example, a function with handle `f` (defined in first line) is used to filter a `find` to return only those objects of type `'andState'`:

```
f = @(h) (strcmp(get(h,'type'), 'andState'));
objArray = thisObject.find('-function', f);
```

`'-isa'` Following value specifies the name of the type of objects to search for. Object types are specified as a string name (e.g., `'Stateflow.State'`, `'Stateflow.Transition'`, etc.) or as a handle to the object type (see method `classhandle`).

`'-method'` Following value is a string that specifies the name of a method belonging to the objects to search for.

`'-not'` No value is paired to this specifier. Instead, this specifier searches for the negative of the following specifier-value pair.

`'-or'` No value is paired to this specifier. Instead, this specifier relates the previous specifier-value pair to the following specifier-value pair in an OR relation.

Note If no logical operator is specified, `-or` is assumed.

`'property'`	The specifier takes on the name of the property. Value is the string value of the specified property for the objects you want to find.
`'-property'`	Following value is the string name of the property that belongs to the objects you want to find.
`'-xor'`	No value is paired to this specifier. Instead, this specifier relates the previous specifier-value pair to the following specifier-value pair in an XOR relation.
`'-regexp'`	No value follows this specifier. Instead, this specifier indicates that the value of the following specifier-value pair contains a regular expression.

Returns

`objArray`	Array of objects found matching the criteria specified (see Arguments)

Example

If a Chart object `c` represents a Stateflow chart, the command `states=c.find('-isa','Stateflow.State')` returns an array, `states`, of all the states in the chart, and the command `states=c.find('Name','A')` returns an array of all objects whose `Name` property is `'A'`.

If state `A`, which is represented by State object `sA`, contains two states, A1 and `A2`, and you specify a `find` command that finds all the states in A as follows,

```
states= sA.find( '-isa','Stateflow.State')
```

then the above command finds three states: `A`, `A1`, and `A2`.

See Also

The methods `findDeep` and `findShallow`

findDeep

Purpose

Return specified objects in this object at all levels of containment below this object

Syntax

```
objArray = thisObject.find(Type)
```

Description

The `findDeep` method of this object returns all objects of the specified type at any and all depths of containment within this object. You specify the object type as a string argument. Only one type is allowed.

Arguments

`thisObject`	The object for which to find objects at all levels of containment. It can be an object of type Root, Machine, Chart, State, Box, Function, or Truth Table.
`Type`	The string name for the type of objects to find. Values can be `'Machine'`, `'Chart'`, `'State'`, `'Box'`, `'Function'`, `TruthTable`, `'Note'`, `'Transition'`, `'Junction'`, `'Event'`, `'Data'`, or `'Target'`.

Returns

`objArray`	Array of objects of the specified type found in this object at any (all) depths.

Example

Chart object `ch` contains state `A` and state `B`, connected by a transition. State `A` contains states `A11` and `A12`, also connected by a transition. State `A11` contains state `A111` and a junction. The command `ch.findDeep('State')` returns an array of state objects for states `A`, `A11`, `A12`, `A111`, and `B`.

See Also

The methods `findShallow` and `find`

Purpose

Return specified objects in this object at the first level of containment below this object

Syntax

```
objArray = thisObject.find(Type)
```

Description

The `findShallow` method of this object returns all objects of the specified type at the first level of containment within this object. You specify the type of object as a string argument. Only one type is allowed.

Arguments

`thisObject`	The object for which to find objects at the first level of containment. It can be an object of type Root, Machine, Chart, State, Box, Function, or Truth Table.
`Type`	The string name of the type of object to find. Value can be `'Machine'`, `'Chart'`, `'State'`, `'Box'`, `'Function'`, `'TruthTable'`, `'Note'`, `'Transition'`, `'Junction'`, `'Event'`, `'Data'`, or `'Target'`. If no type is specified (i.e., no argument), all possible object types are found.

Returns

`objArray`	Array of objects of the specified type found in this object at the top level of containment (i.e., first depth)

Example

Chart object `ch` contains states `A` and state `B`, connected by a transition. State A contains states `A11` and `A12`, also connected by a transition. State `A11` contains state `A111` and a junction. The command `ch.findShallow ('State')` returns an array of state objects for states `A` and `B`.

See Also

The methods `findDeep` and `find`

generate

Purpose

Generate code incrementally for this target

Syntax

```
thisTarget.generate
```

Description

The `generate` method generates code incrementally for this target. If a complete code generation has already taken place, it performs an incremental generation for only those portions of the target corresponding to charts that have changed since the last code generation.

Arguments

`thisTarget` The target for which to generate code.

Returns

None

Example

If `t` is a Target object, the command `t.generate` generates code for the Stateflow charts that have changed in the target's model since the last code generation.

See Also

The methods `build`, `rebuildAll`, `regenerateAll`, and `make`

Purpose

Return a MATLAB structure containing the property settings of this object or an array of objects

Syntax

```
propList = thisObject.get(prop)
```

Description

The `get` method returns and displays a MATLAB structure containing the settings for the specified property of this object. If no property is specified, the settings for all properties are returned.

The `get` method is also vectorized so that it returns an `m`-by-`n` cell array of values for an array of `m` objects and an array of `n` properties.

Arguments

`thisObject`	The object for which to get specified property.
`prop`	String name of property (e.g., `'FontSize'`) to get value for. Can also be an array of properties (see return `propList` below). If no property is specified, a list of all properties is returned.

Returns

`propList`	MATLAB structure listing the properties of this object. Can also be an `m` by `n` cell array of values if `thisObject` is an array of `m` objects and `prop` is an array of `n` properties.

Example

State `A` is represented by the State object `sA`.

The following command lists the properties of state `A`:

```
sA.get
```

The following command returns a handle to a MATLAB structure of the properties of state `A` to the workspace variable `Aprops`:

```
Aprops = sA.get
```

getCodeFlag

Purpose

Return the specified code flag

Syntax

```
thisTarget.getCodeFlag(name)
```

Description

The `getCodeFlag` method returns the value of a particular code flag whose name you specify.

Arguments

`thisTarget`	The target for which to get code flag value
`name`	The short string name of the code flag for which to get value. See "CodeFlagsInfo Property of Targets" on page 15-60 for a list of these names.

Returns

None

Example

Assume that the Target object `x` represents the simulation target `sfun` for the loaded model. If `m` is the Stateflow machine object for this model, you can obtain `x` with the following command:

```
x = m.find('-isa','Stateflow.Target','-and', 'Name','sfun')
```

The simulation target has two code flags: `debug` and `echo`. You can verify this by looking at the `CodeFlagsInfo` property of `x`. See the description of this property in the reference section "Target Properties" on page 15-59 for more information.

In the Stateflow user interface the `debug` code flag is enabled or disabled through the **Enable debugging/animation** check box in the **Coder Options** dialog. By default, this flag is turned on for the simulation target. You can verify this with the following command:

```
t.getCodeFlag('debug')
```

Similarly, you can check the value of the `echo` code flag, which is enabled or disabled through the **Echo expressions without semicolons** check box of the same dialog, with the following command:

```
t.getCodeFlag('echo')
```

See Also

The method `setCodeFlag`

Purpose

Display the list of properties for this object along with accompanying descriptions

Syntax

```
thisObject.help
```

Description

The `help` method returns a list of properties for any object. To the right of this list appear simple descriptions for each property. Some properties do not have descriptions because their names are descriptive in themselves.

Arguments

None

Returns

None

Example

If `j` is an API handle to a Stateflow junction, the command `j.help` returns a list of the property names and descriptions for a Stateflow API object of type Junction.

innerTransitions

Purpose

Return the inner transitions that originate with this chart or state and terminate on a contained object

Syntax

```
transitions = thisObject.innerTransitions
```

Description

The `innerTransitions` method returns the inner transitions that originate with this object and terminate on a contained object.

Arguments

None

Returns

`thisObject` Object for which to get inner transitions. Can be of type State, Box, or Function.

`transitions` Array of inner transitions originating with this object and terminating on a contained state or junction.

Example

State `A` contains state `A1`, and state `A1` contains state `A11`. State `A` has two transitions, each originating from its inside edge and terminating inside it. These are inner transitions. One transition terminates with state `A1` and the other terminates with state `A11`. The `innerTransitions` method of state `A` returns both of these transitions.

Purpose

Incrementally compile this target with no code generation

Syntax

```
thisTarget.make
```

Description

For a simulation target (`sfun`) a dynamic link library (`sfun.dll`) is compiled from the generated code. The `make` method performs an incremental compile of this target with no code generation. It performs the compile for only those portions of generated code that have changed since the last compile.

Arguments

`thisTarget` The target for which to do make

Returns

None

Example

If `t` is a Target object, the command `t.make` incrementally compiles generated code for that target. If `t` is a simulation target (`sfun`), its compiled code is then linked and loaded into the target `.dll`. file.

See Also

The methods `build`, `rebuildAll`, `generate`, and `regenerateAll`

methods

Purpose

List the names of the methods belonging to this object

Syntax

```
thisObject.methods
```

Description

The `methods` method lists the names of the methods belonging to this object.

Note The `methods` method for this object displays some internal methods that are not applicable to Stateflow use, and are not documented. These are as follows: `areChildrenOrdered`, `getChildren`, `getDialogInterface`, `getDialogSchema`, `getDisplayClass`, `getDisplayIcon`, `getDisplayLabel`, `getFullName`, `getHierarchicalChildren`, `getPreferredProperties`, `isHierarchical`, `isLibrary`, `isLinked`, `isMasked`.

Arguments

`thisObject`	Object for which to list methods. Can be of any Stateflow object type.

Returns

None

Example

If state `A` is represented by State object `sA`, the command `sA.methods` lists the methods of state `A`.

Purpose

Return an array of outer transitions for this object

Syntax

```
transitions = thisObject.outerTransitions
```

Description

The `outerTransitions` method returns an array of transitions that exit the outer edge of this object and terminate on objects outside the containment of this object.

Arguments

None

Returns

`thisObject` The object for which to find outer transitions. Can be of object type State, Box, or Function.

`transitions` An array of transitions exiting the outer edge of this state.

Example

A chart contains three states, `A`, `B`, and `C`. State `A` is connected to state `B` through a transition from state `A` to state `B`. State B is connected to state `C` through a transition from state `B` to state `C`. And state `C` is connected to state `A` through a transition from state `C` to state `A`. If state `A` is represented by State object handle `sA`, the command `sA.outerTransitions` returns the transition from state `A` to state `B`.

outputData

Purpose

Create, retrieve, or delete a data output to Simulink of this state's activity status

Syntax

```
StateData = thisState.outputData (action)
```

Description

The `outputData` method of this state creates, retrieves, or deletes a special data object of type `State`. This data is attached internally to an output port on this state's Stateflow block in Simulink to output the activity status of this state to Simulink during run-time.

Note You cannot use the Stateflow Explorer to create Data objects of type `State`.

Arguments

`thisState` The state object for which to add a special port.

`action` This string value can be one of the following:

- `'create'` — Returns a new data object of type `State` and attaches it internally to a new state activity output port on this state's Stateflow block.
- `'get'` — Returns this state's existing data object of type `State` attached internally to an existing state activity output port on this state's Stateflow block.
- `'delete'` — Deletes this state's data object of type `State` and the state activity output port on its Stateflow block to which it is attached.

Returns

`StateData` The data object of type `State` for this state

Example

If state `A` is represented by State object `sA`, the following command creates a new data object of type State, which is output to Simulink and contains state `A`'s activity:

```
s.outputData('create')
```

The Stateflow Chart block in Simulink that contains state `A` now has an output port labeled `A`, the name of state `A`. In Explorer, state `A` now contains a data object of type `State` whose scope is `Output to Simulink`.

The following command returns a Data object, `d`, for the data output to Simulink containing state `A`'s activity:

```
s.outputData('get')
```

The following command deletes the data output to Simulink containing state `A`'s activity:

```
s.outputData('delete')
```

parse

Purpose

For Chart objects, parse this chart; for Machine objects, parse the charts in this machine

Syntax

```
thisChart.parse
thisMachine.parse
```

Description

For Chart objects, the `parse` method parses this chart. This is equivalent to selecting **Parse** from the **Tools** menu of the Stateflow diagram editor for this chart.

For Machine objects, the `parse` method parses all the charts in this machine.

Arguments

`thisChart`	The chart to parse
`thisMachine`	The machine containing charts to parse

Returns

None

Example

If `ch` is a handle to an API object representing a chart, then the command `ch.parse` parses the chart.

Purpose

Paste the objects in the Clipboard to the specified container object

Syntax

```
clipboard.pasteTo(newContainer)
```

Description

The `paste` method pastes the contents of the Clipboard to the specified container object. The receiving container is specified through a single argument. Use of this method assumes that you placed objects in the Clipboard with the `copy` method.

Arguments

`newContainer`	The Stateflow object to receive a copy of the contents of the Clipboard object. If the objects in the Clipboard are all graphical (states, boxes, functions, notes, transitions, junctions), this object must be a chart or subchart.

Returns

None

Example

See the section “Copying Objects” on page 13-29.

rebuildAll

Purpose

Completely rebuild this target

Syntax

```
thisTarget.rebuildAll
```

Description

The `rebuildAll` method completely rebuilds this target with the following actions:

- Parses all charts completely.
- Regenerates code for all charts completely.
- For a simulation target (`sfun`), a dynamic link library (`sfun.dll`) is compiled from the generated code.

Arguments

`thisTarget` The Stateflow target to rebuild

Returns

None

Example

If `t` is a Target object, the command `t.rebuildAll` completely rebuilds that target.

See Also

The methods `build`, `generate`, `regenerateAll`, and `make`

Purpose

Completely regenerate code for this target

Syntax

```
thisTarget.regenerateAll
```

Description

The `regenerateAll` method regenerates this target. Regardless of previous code generations, it regenerates code for all charts in this target's model.

Arguments

`thisTarget` The Stateflow target for which to regenerate code

Returns

None

Example

If `t` is a Target object, the command `t.regenerateAll` completely regenerates code for the Stateflow charts in that target's model.

See Also

The methods `build`, `rebuildAll`, `generate`, and `make`

set

Purpose

Set specified properties with the specified values

Syntax

```
thisObject.set(propName,value,...)
```

Note Arguments can consist of an indefinite number of property (name, value) pairs.

Description

The `set` method sets the value of a specified property or sets the values of a set of specified properties for this object. You specify properties and values through pairs of property (name, value) arguments.

The `get` method is also vectorized so that it sets an `m`-by-`n` cell array of values for an array of `m` objects and an array of `n` properties.

Arguments

thisObject	The object for which the specified property is set. Can be any Stateflow object.
`propName`	String name of the property to set (e.g., `'FontSize'`). Can also be a cell array of `m` property names.
`value`	New value for the specified property. Can be a cell array of `m`-by-`n` values if `thisObject` is an array of `m` objects and `propName` is an array of `n` property names.

Returns

None

Example

The following command sets the `Name` and `Description` properties of the State object `s`:

```
s.set('Name', 'Kentucky', 'Description', 'Bluegrass State')
```

The following command sets the `Position` property of the State object `s`:

```
s.set('Position',[200,119,90,60])
```

Purpose

Set the specified code flag to the value you specify

Syntax

```
thisTarget.setCodeFlag(name,value)
```

Description

The `setCodeFlag` method sets the value of a code flag whose name you specify.

Arguments

thisTarget	Target object for which to set code flag.
`name`	String name of code flag. See "CodeFlagsInfo Property of Targets" on page 15-60 for a list of these names.
`value`	Value of code flag. Can be of any type.

Flag values can vary in type. Use the property `CodeFlagsInfo` to obtain the type for a particular flag.

Returns

None

Example

Assume that the Target object `x` represents the simulation target `sfun` for the loaded model. If `m` is the Stateflow machine object for this model, you can obtain `x` with the following command:

```
x = m.find('-isa','Stateflow.Target','-and', 'Name','sfun')
```

The simulation target has two code flags: `debug` and `echo`. You can verify this by looking at the `CodeFlagsInfo` property of `x` with the following command:

```
x.CodeFlagsInfo.name
```

In the Stateflow user interface the `debug` code flag is enabled or disabled through the **Enable debugging/animation** check box in the **Coder Options** dialog. By default, this flag is turned on (==1) for the simulation target, which you can verify with the following command:

```
t.getCodeFlag('debug')
```

If you want to disable debugging, enter the following command:

```
t.setCodeFlag('debug',0)
```

See Also

The method `getCodeFlag`

sfclipboard

Purpose

Return a handle to the Clipboard object

Syntax

```
cb = sfclipboard
```

Description

The global method `sfclipboard` method returns a handle to the Clipboard object. There is only one Clipboard object. Use it to copy objects from one owner to another. See the section “Copying Objects” on page 13-29.

Arguments

None

Returns

`cb`	Handle to the Clipboard object

Purpose

Close all Simulink models containing Stateflow diagrams and exit the Stateflow environment

Syntax

```
sfexit
```

Description

The global `sfexit` method closes all Simulink models containing Stateflow diagrams and exits the Stateflow environment.

Caution Using the `sfexit` method causes all models in Simulink to close without a request to save them.

Arguments

None

Returns

None

sfhelp

Purpose Display Stateflow online help

Syntax `sfhelp`

Description The global `sfhelp` method displays Stateflow online help.

Arguments None

Returns None

Purpose

Create a Simulink model containing an empty Stateflow block

Syntax

```
sfnew ModelName
```

Description

The `sfnew` method creates and displays an untitled Simulink model containing an empty Stateflow block. If you pass the `sfnew` method a model name it creates a Simulink model with the title you specified.

Arguments

`modelName` Optional string model name to save model to

Returns

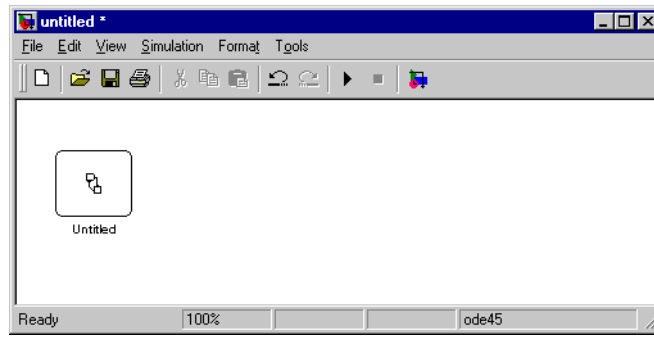
None

Example

Create an untitled Simulink model that contains an empty Stateflow block.

```
sfnew
```

The new model appears as follows:



sfprint

Purpose

Display the visible portion of a Stateflow diagram

Syntax

```
sfprint (objects, format, outputOption, printEntireChart)
```

Description

The `sfprint` method prints the visible portion of the specified Stateflow diagram. A read-only preview window appears while the print operation is in progress. An informational message box appears indicating that the printing operation is starting.

See “Printing the Current Stateflow Diagram” on page 5-90 for information on printing Stateflow diagrams that are larger than the editor display area.

Arguments

`objects`

String name of a chart, model, system, or block. For a model or system, all member diagrams are printed.

-or-

Integer `Id` of a chart

Can also be a cell array of any combination of the preceding objects or a vector of `Id`s.

`format`

Optional string specification of the destination for the visible portion of the specified Stateflow diagram, as follows:

- `'default'`— Default printer
- `'ps'`— PostScript file
- `'psc'` — Color PostScript file
- `'eps'` — Encapsulated PostScript file
- `'epsc'` — Color Encapsulated PostScript file
- `'tif'` — TIFF file.
- `'jpg'` — JPEG file.
- `'png'` — PNG file.
- `'meta'` — Save to the clipboard as a meta file (PC version only).
- ‘`'bitmap'` — Save to the clipboard as a bitmap file (PC version only).

If the format parameter is absent, the default format is `'ps'` and output is directed to the default printer.

`outputOption` Optional string specification of output with the following options:

- '*`FileName`*' — Entered name of file to write to. This file is overwritten if more than one chart is printed.
- `'promptForFile'` — Filenames requested by prompt.
- `'printer'` — Output is sent to the default printer. Use only with `'default'`, `'ps'`, or `'eps'` formats.
- `'file'` — Output is sent to a default file.
- `'clipboard'` — Output is copied to the clipboard.

If nothing is specified, output is sent to the default printer.

`printEntireChart` Optional Boolean argument with the following two possible values:

- 1 — Print the entire chart (default).
- 0 — Print the current view of the specified charts.

Returns

None

Example

Assume that you have a Simulink model named `myModel` loaded into MATLAB that has two charts named `Chart1` and `Chart2`. Further, both `Chart1` and `Chart2` are represented by the Stateflow API Chart objects ch1 and ch2, respectively.

Command	Result
`sfprint('myModel')`	Prints the visible portions of both `Chart1` and `Chart2` to the default printer.
`sfprint('myModel','ps')`	Prints the visible portion of both `Chart1` and `Chart2` to a PostScript file.

Command	Result
`sfprint(ch1.Id,'psc')`	Prints the visible portion of `Chart1` to a color PostScript file.
`sfprint([ch1.Id, ch2.Id])`	Prints the visible portions of both `Chart1` and `Chart2` to the default printer.

sfroot

Purpose

Return a handle to the Root object

Syntax

```
rt = sfroot
```

Description

The global method `sfroot` returns a handle to the Root object. There is only one Root object. It is the top object in the Stateflow model hierarchy of objects. Use it to begin accessing your Stateflow charts. See the section "Access the Machine Object" on page 13-9.

Arguments

None

Returns

`rt` Handle to the Root object

Purpose

Save the current state machine (Simulink model)

Syntax

```
sfsave
sfsave (modelname)
sfsave ('defaults')
sfsave (machine.Id, 'saveasname')
```

Description

The global function `sfsave` saves the current model or a specified model either with the current model name or with an assigned “save as” model name.

Arguments

`modelname`	Optional string name of the current model to save. If no model is specified, saves the currently loaded model. Maximum size is 25 characters.
`'defaults'`	If the string literal `'defaults'` is specified, saves the current environment default settings in the defaults file.
`machine.Id`	Integer property `Id` of the Machine object representing the machine (model) you want to save.
`saveasname`	Optional string model name to save current model to. If this is not specified, the specified model name is saved under its own name. Maximum size is 25 characters.

Returns

None

Example

Assume that you have a new Stateflow model. Its current default name is `'untitled'` and `m` is a handle to its Machine object.

- The command `sfsave` saves the current model under the name `'untitled'` in the work directory.
- The command `sfsave('untitled')` saves the currently loaded model named `untitled` under the name `untitled` in the work directory.
- The command `sfsave (m.Id,'myModel')` saves the model as the model `myModel`.
- The command `sfsave ('defaults')` saves the current environment default settings in the defaults file.

sfversion

Purpose

Return the current version of Stateflow

Syntax

```
ver = sfsave (specifier)
```

Description

The global function `sfversion` returns the current Stateflow version. Output is based on an options specifier, which is described below in the Arguments section.

Arguments

(nothing)	Returns the current Stateflow version in a readable string format with time-stamp.
'STRING'	Returns the current Stateflow version as a dot-delimited string.
'NUMBER'	Returns the current Stateflow version as a `double`.

'FULL_STRING'	Returns the full current Stateflow version as a dot-delimited string of numbers in the following 14-digit format:
	`sfr.matr.matpr.fr.build`
	where
	• `sfr` is the Stateflow release number in the following format:
	`<major release>.<point release>.<bug release>`
	For example, 4.2.1.
	• `matr` is the MATLAB release, for example, 11 for R11.
	• `matpr` is the MATLAB point release, for example, 03 for Beta 3).
	• `fr` is the final release flag.
	This digit is 1 for an official final release and 0 otherwise.
	• `build` is a 6-digit internal build number.

'FULL_NUMBER'	Returns the current Stateflow full version number as a double.

Returns

`ver`	String or double of current Stateflow version, depending upon specifier argument

Example

Assume that you have just loaded the Beta 2 Release 13 of MATLAB with Version 5.0 of Stateflow. The date is July 14, 2002, and the time is 3:02 p.m. In this case, you might receive output like the following:

- The command `sfversion` returns the following:
  Version 5.0 Beta 2 (R13) dated July 14 2002, 15:02:34
- The command `sfversion('STRING')` returns the following:
  5.0.0
- The command `sfversion('number')` returns the following:
  5
- The command `sfversion('full_string')` returns the following:
  5.0.0.13.00.0.000001
- The command `sfversion('full_number')` returns the following:
  5.0013e+007

sourcedTransitions

Purpose

Return the transitions that have this object as their source

Syntax

```
transitions = thisObject.sourcedTransitions
```

Description

The `sourcedTransitions` method returns all inner and outer transitions that have their source in this object.

Arguments

`transitions` The source object of the returned transitions. Can be of type State, Box, Function, or Junction.

Returns

`transitions` Array of all transitions whose source is this object

Example

Suppose that a chart contains three states, `A`, `B`, and state `A1`, which is contained by state `A`. The chart also has three transitions: one from `A` to `B` labeled `AtoB`, one from `B` to `A` labeled `BtoA`, and one from the inner edge of `A` to its state `A1` (inner transition) labeled `AtoA1`. If State object `sA` represents state `A`, the command `sA.sourcedTransitions` returns two transitions: the outer transition labeled `AtoB` and the inner transition labeled `AtoA1`.

Purpose

Open the Stateflow model window

Syntax

```
stateflow
```

Description

The global `stateflow` method opens the Stateflow model window. The model contains an untitled Stateflow block, an Examples block, and a manual switch. The Stateflow block is a masked Simulink model and is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.

Every Stateflow block has a corresponding S-function target called `sfun`. This target is the agent Simulink interacts with for simulation and analysis.

Arguments

None

Returns

None

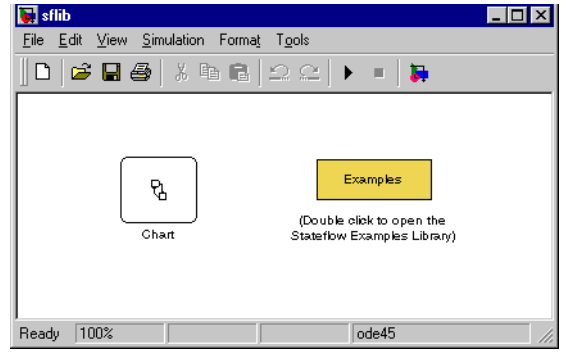
Example

This example shows how to open the Stateflow model window and use a Stateflow block to create a Simulink model:

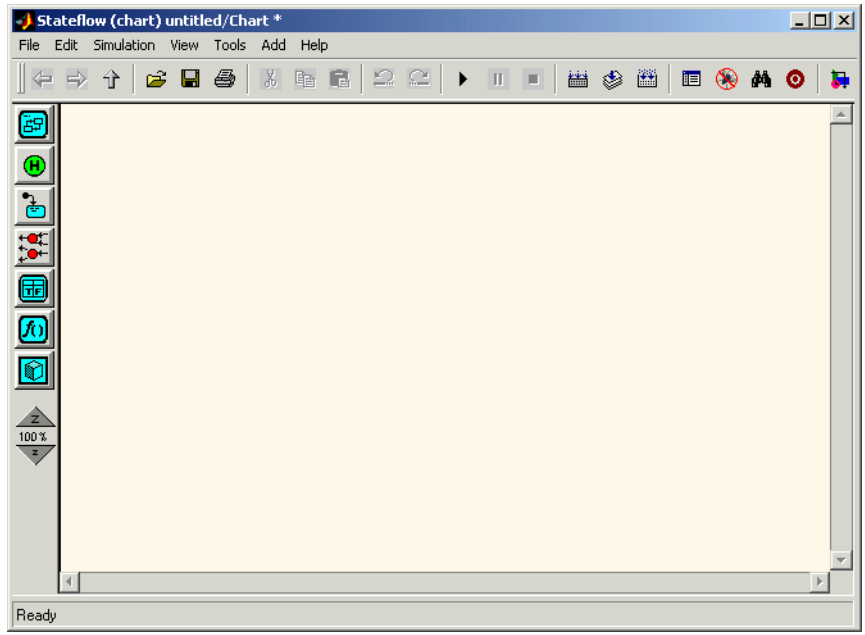
1. Invoke Stateflow.

   ```
   stateflow
   ```

   The Stateflow model window and an untitled Simulink model containing a Stateflow block are displayed.



2. Double-click the untitled Stateflow block in the untitled Simulink model to invoke a Stateflow editor window.



3 Create the underlying Stateflow diagram.

Purpose

Constructor for creating a box

Syntax

```
box_new = Stateflow.Box(parent)
```

Description

The `Stateflow.Box` method is a constructor method for creating boxes in a parent chart, state, box, or function, that returns a handle to an Event object for the new function.

Arguments

`parent`	Handle to an object for the parent chart, state, box, or function of the new box

Returns

`box_new`	Handle to the Box object for the new box

Example

If `sA` is a handle to a State object for an existing state `A`, the following command creates a new box parented (contained by) state `A`:

```
box_new = Stateflow.Box(sA)
```

The new box is unnamed and appears in the upper left hand corner inside state `A`. `box_new` is a handle to a Box object for the new box.

Stateflow.Data

Purpose

Constructor for creating a data

Syntax

```
data_new = Stateflow.Data(parent)
```

Description

The `Stateflow.Data` method is a constructor method for creating data for a parent machine, chart, state, box, or function, that returns a handle to the Data object for the new data.

Arguments

`parent`	Handle to an object for the parent machine, chart, state, box, or function of the new data

Returns

`data_new`	Handle to the Data object for the new data

Example

If `sA` is a handle to a State object for an existing state `A`, the following command creates a new data parented (contained by) state `A`:

```
data_new = Stateflow.Data(sA)
```

The new data is named `'data'` with an incremented integer suffix to distinguish additional creations. `data_new` is a handle to the Data object for the new data.

Purpose

Constructor for creating an event

Syntax

```
event_new = Stateflow.Event(parent)
```

Description

The `Stateflow.Event` method is a constructor method for creating an event for a parent machine, chart, state, box, or function, that returns a handle to an Event object for the new event.

Arguments

`parent`	Handle to parent machine, chart, state, box, or function of new event

Returns

`event_new`	Handle to the Event object for the new event

Example

If `sA` is a handle to a State object for an existing state `A`, the following command creates a new event parented (contained by) state `A`:

```
event_new = Stateflow.Event(sA)
```

The new event is named `'event'` with an incremented suffix to distinguish additional creations . `event_new` is a handle to an Event object for the new event that you use to rename the event, set its properties, and execute Event methods for the event.

Stateflow.Function

Purpose

Constructor for creating a function

Syntax

```
function_new = Stateflow.Function(parent)
```

Description

The `Stateflow.Function` method is a constructor method for creating functions in a parent chart, state, box, or function, that returns a handle to a Function object for the new function.

Arguments

`parent`	Handle to parent chart or state of the new function

Returns

`function_new`	Handle to a Function object for the new function

Example

If `sA` is a handle to a State object for the existing state `A`, the following command creates a new function parented (contained by) state `A`:

```
function_new = Stateflow.Function(sA)
```

The new function is unnamed and appears in the upper left corner inside of state `A` in the diagram editor. `function_new` is a handle to the Function object for the new function that you use to rename the function, set its properties, and execute its methods.

Purpose

Constructor for creating a junction

Syntax

```
junc_new = Stateflow.Junction(parent)
```

Description

The `Stateflow.Junction` method is a constructor method for creating a junction in a parent chart, state, box, or function, that returns a handle to the Junction object for the new junction.

Arguments

`parent`	Handle to the object for the parent chart, state, box, or function or the new junction

Returns

`junc_new`	Handle to the Junction object for new junction

Example

If `sA` is a handle to a State object for the existing state `A`, the following command creates a new junction parented (contained by) state `A`:

```
junc_new = Stateflow.Junction(sA)
```

The new junction appears in the middle of state `A` in the diagram editor. `junc_new` is a handle to the Junction object for the new junction that you use to set its properties, and execute its methods.

Stateflow.Note

Purpose

Constructor for creating a note

Syntax

```
note_new = Stateflow.Note(parent)
```

Description

The `Stateflow.Note` method is a constructor method for creating notes for a parent chart, state, box, or function, that returns a handle to the Note object for the new note.

Arguments

`parent`	Handle to the object for the parent chart, or subchart for the new note

Returns

`note_new`	Handle to the Note object for the newly created note

Example

If `sA` is a handle to a State object for the existing state `A`, the following command creates a new note parented (contained by) state `A`:

```
note_new = Stateflow.Note(sA)
```

The new note is placed in the upper left hand corner of state `A` in the diagram editor, but is invisible because it has no text content. `note_new` is a handle to the Note object for the new note, that you use to set its text content with a command like the following:

```
note_new.Text = 'This is a note'
```

Purpose

Constructor for creating a state

Syntax

```
state_new = Stateflow.State(parent)
```

Description

The `Stateflow.State` method is a constructor method for creating a state for a parent chart, state, box, or function, that returns a handle the State object for the new state.

Arguments

`parent`	Handle to the object for the parent chart, state, box, or function for the new state

Returns

`state_new`	Handle to State object for newly created state

Example

If `sA` is a handle to a State object for the existing state `A`, the following command creates a new state parented (contained by) state `A`:

```
state_new = Stateflow.State(sA)
```

The new state appears in the upper left hand corner of state `A` in the diagram editor. `state_new` is a handle to the State object for the new state that you use to rename the state, set its properties, and execute its methods.

Stateflow.Target

Purpose

Constructor for creating a target

Syntax

```
target_new = Stateflow.Target(parent_m)
```

Description

The `Stateflow.Target` method is a constructor method for creating a target for a parent machine, that returns a handle to the Target object for the new target.

Arguments

`parent_m` Handle to object for the parent machine of the new target

Returns

`target_new` Handle to the Target object for the newly created target

Example

The following command creates a new target for the machine with the Machine object whose handle is `pm`:

```
target_new = Stateflow.Target(pm)
```

The preceding command creates a custom target with name `untitled`. `target_new` is a handle to the Target object of the new target which you can use to rename and set properties for the target. The following command renames the new target to `rtw`, thus making it the Real-Time Workshop (RTW) target for its parent machine:

```
target_new.Name = 'rtw'
```

Purpose

Constructor for creating a truth table

Syntax

```
truth_table_new = Stateflow.TruthTable(parent)
```

Description

The `Stateflow.TruthTable` method is a constructor method for creating truth tables in a parent chart, state, box, or function, that returns a handle to a Truth Table object for the new truth table.

Arguments

`parent`	Handle to parent chart or state of new truth table

Returns

`truth_table_new`	Handle to Truth Table object for new truth table

Example

If `sA` is a handle to a State object for the existing state `A`, the following command creates a new truth table parented (contained by) state `A`:

```
truth_table_new = Stateflow.TruthTable(sA)
```

The new truth table is unnamed and appears in the upper left corner inside of state `A` in the diagram editor. `truth_table_new` is a handle to the Truth Table object for the new truth table that you use to rename it, set its properties, and execute its methods.

struct

Purpose

Return a MATLAB structure containing the property settings of this object

Syntax

```
propList = thisObject.struct
```

Description

The `struct` method returns and displays a MATLAB structure containing the property settings of this object.

Note You can change the values of the properties in this structure just as you would a property of the object. However, the MATLAB structure is not a Stateflow object and changing it does not affect the Stateflow model.

Arguments

`transitions`	The object for which to display property settings. Can be any Stateflow object type.

Returns

`propList`	MATLAB structure listing the properties of this object

Example

If State object `sA` represents a state `A`, the command `x = sA.struct` returns a MATLAB structure `x`. You can use dot notation on `x` to report properties or set the values of other variables. For example, the command `y=x.Name` sets the MATLAB variable `y` to the value of the `Name` property of state `A`, which is `'A'`. The command `x.Name = 'Kansas'` sets the `Name` property of `x` to `'Kansas'` but does not change the `Name` property of state `A`.

Purpose

Make this object visible for editing

Syntax

```
thisObject.view
```

Description

The `view` method opens the object in its appropriate editing environment as follows:

- For Chart objects, the `view` method opens the chart in a diagram editor, if it is not already open, and brings it to the foreground.
- For State, Box, Function, Note, Junction, and Transition objects, the `view` method does the following:
  - **a** Opens the chart containing the object in a diagram editor if it is not already open.
  - **b** Highlights the object.
  - **c** Zooms the object's diagram editor to the level of full expanse of the object's containing state or chart.
  - **d** Brings the diagram editor for this object to the foreground.
- For Truth Table objects, the `view` method opens the truth table editor for this truth table:
- For Event, Data, and Target objects, the `view` method opens the Explorer window.

Arguments

`thisObject`	Object for which to display editing environment. Can be an object of type Chart, State, Box, Function, Truth Table, Note, Transition, Junction, Event, Data, or Trigger.

Returns

None

zoomIn and zoomOut

Purpose

Zoom in or out on this chart

Syntax

```
thisChart.zoomIn
thisChart.zoomOut
```

Description

The methods `zoomIn` and `zoomOut` cause the Stateflow diagram editor window for this chart to zoom in or zoom out, respectively, by 20 percentage points.

Note The `zoomIn` and `zoomOut` methods do not open or give focus to the Stateflow diagram editor for this chart.

Arguments

`thisChart` Chart object to zoom in or out on.

Returns

None

Example

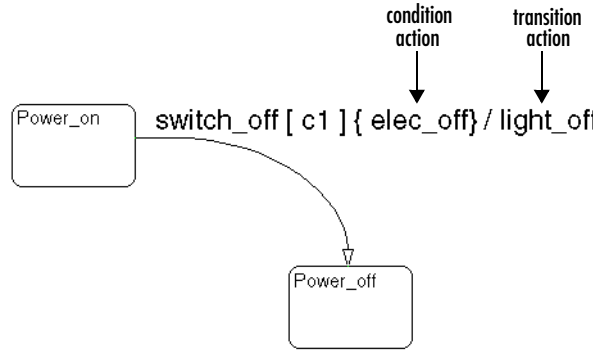
If the Chart object `ch` represents a Stateflow chart at the zoom level of 100%, the command `ch.zoomIn` raises the zoom level to 120%.

A

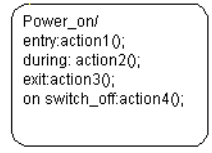
Glossary

actions

Actions take place as a part of Stateflow diagram execution. The action can be executed as part of a transition from one state to another, or depending on the activity status of a state. Transitions can have condition actions and transition actions. For example,



Action language defines the categories of actions you can specify and their associated notations. For example, states can have `entry`, `during`, `exit`, and `on` *`event_name`* actions as shown by the following:



An action can be a function call, a broadcast event, a variable assignment, and so on. For more information on actions and action language, see Chapter 7, “Action Language.”

API (application programming interface)

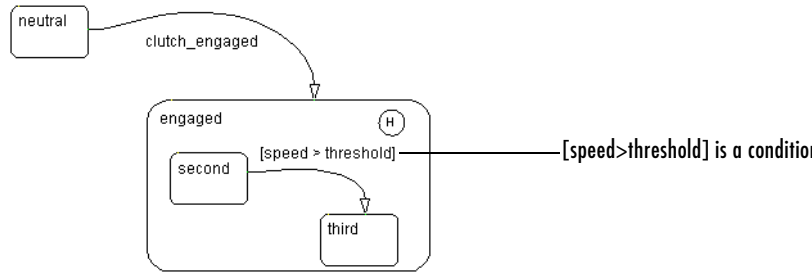
Format provided to access and communicate with an application program from a programming or script environment.

chart instance

Link from a Stateflow model to a chart stored in a Simulink library. A chart in a library can have many chart instances. Updating the chart in the library automatically updates all the instances of that chart.

condition

Boolean expression to specify that a transition occurs if the specified expression is true. For example,

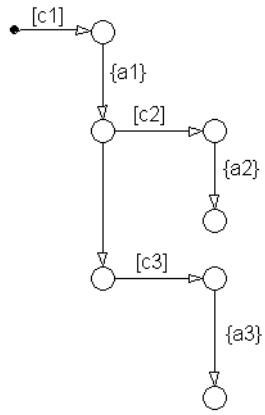


The action language defines the notation to define conditions associated with transitions. See “Action Language” on page 3-42 for more information.

connective junction

Decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior.

This example shows how connective junctions (displayed as small circles) are used to represent the decision flow of an `if` code structure.




```
if [c1]{
     a1
     if [c2]{
        a2
     }else if [c3]{
       a3
     }
}
```

Name	Button Icon	Description
Connective junction		One use of a connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.

See “Connective Junctions” on page 3-31 for more information.

data *Data* objects store numerical values for reference in the Stateflow diagram.

See “Defining Data” on page 6-14 for more information on representing data objects.

data dictionary Database where Stateflow diagram information is stored. When you create Stateflow diagram objects, the information about those objects is stored in the data dictionary once you save the Stateflow diagram.

Debugger See “Stateflow Debugger” on page A-10.

decomposition

A state has a *decomposition* when it consists of one or more substates. A Stateflow diagram that contains at least one state also has decomposition. Representing hierarchy necessitates some rules around how states can be grouped in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level in the hierarchy must be of the same decomposition.

Parallel (AND) State Decomposition. *Parallel (AND) state decomposition* is indicated when states have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are active at the same time. The activity within parallel states is essentially independent.

Exclusive (OR) State Decomposition. *Exclusive (OR) state decomposition* is represented by states with solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. Only one state at the same level in the hierarchy can be active at a time.

default transition

Primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information. Default transitions are also used to specify that a junction should be entered by default. A default transition is represented by selecting the default transition object from the toolbar and then dropping it to attach to a destination object. The default transition object is a transition with a destination but no source object.

Name	Button Icon	Description
Default transition		Use a default transition to indicate, when entering this level in the hierarchy, which state becomes active by default.

See “Default Transitions” on page 3-26 for more information.

events — *Events* drive the Stateflow diagram execution. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur and/or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are added, removed, and edited through the Stateflow Explorer. See "Defining Events" on page 6-2 for more information.

Explorer — A tool for displaying, modifying, and creating data and event objects for any parent object in Stateflow. The Explorer also displays, modifies, and creates targets for the Stateflow machine. See "Stateflow Explorer" on page A-10.

Finder — A tool to search for objects in Stateflow diagrams on platforms that do not support the Simulink Find tool. See "Stateflow Finder" on page A-10.

finite state machine (FSM) — Representation of an event-driven system. FSMs are also used to describe reactive systems. In an event-driven or reactive system, the system transitions from one mode or state to another prescribed mode or state, provided that the condition defining the change is true.

flow graph — Set of decision flow paths that start from a transition segment that, in turn, starts from a state or a default transition segment.

flow path — Ordered sequence of transition segments and junctions where each succeeding segment starts on the junction that terminated the previous segment.

flow subgraph — Set of decision flow paths that start on the same transition segment.

graphical function — Function whose logic is defined by a flow graph. See "Using Graphical Functions in Stateflow Charts" on page 5-48.

hierarchy — *Hierarchy* enables you to organize complex systems by placing states within other higher-level states. A hierarchical design usually reduces the number of transitions and produces neat, more manageable diagrams. See "Stateflow Hierarchy of Objects" on page 2-21 for more information.

history junction — Provides the means to specify the destination substate of a transition based on historical information. If a superstate has a history junction, the transition to the destination substate is defined to be the substate that was most recently

visited. The history junction applies to the level of the hierarchy in which it appears.

Name	Button Icon	Description
History junction		Use a history junction to indicate, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.

See these sections for more information:

- "History Junctions" on page 3-38
- "Default Transition and a History Junction Example" on page 4-37
- "Labeled Default Transitions Example" on page 4-39
- "Inner Transition to a History Junction Example" on page 4-48

inner transitions Transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with XOR decomposition. Use of inner transitions can greatly simplify a Stateflow diagram.

See "Inner Transitions" on page 3-22 and "Inner Transition to a History Junction Example" on page 4-48 for more information.

library link Link to a chart that is stored in a library model in a Simulink block library.

library model Stateflow model that is stored in a Simulink library. You can include charts from a library in your model by copying them. When you copy a chart from a library into your model, Stateflow does not physically include the chart in your model. Instead, it creates a link to the library chart. You can create multiple links to a single chart. Each link is called a *chart instance*. When you include a chart from a library in your model, you also include its Stateflow machine. Thus, a Stateflow model that includes links to library charts has multiple Stateflow machines. When Stateflow simulates a model that includes charts from a library model, it includes all charts from the library model even if there are links to only some of its models. However, when Stateflow generates a stand-alone or RTW target, it includes only those charts for which there are links. A model that includes links to a library model can be simulated only if all charts in the library model are free of parse and compile errors.

machine Collection of all Stateflow blocks defined by a Simulink model. This excludes chart instances from library links. If a model includes any library links, it also includes the Stateflow machines defined by the models from which the links originate.

notation Defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects

parallelism *A* system with *parallelism* can have two or more states that can be active at the same time. The activity of parallel states is essentially independent. Parallelism is represented with a parallel (AND) state decomposition.

See "State Decomposition" on page 3-9 for more information.

Real-Time Workshop Automatic C language code generator for Simulink. It produces C code directly from Simulink block diagram models and automatically builds programs that can be run in real time in a variety of environments. See the Real-Time Workshop documentation for more information.

rtw target Executable built from code generated by the Real-Time Workshop. See Chapter 11, "Building Targets," for more information.

S-function When using Simulink together with Stateflow for simulation, Stateflow generates an *S-function* (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the `sfun` target within Stateflow.

For more information, see Using Simulink documentation.

semantics *Semantics* describe how the notation is interpreted and implemented behind the scenes. A completed Stateflow diagram communicates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Simulink

*S*oftware package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

It allows you to represent systems as block diagrams that you build using your mouse to connect blocks and your keyboard to edit block parameters. Stateflow is part of this environment. The Stateflow block is a masked Simulink model. Stateflow builds an S-function that corresponds to each Stateflow machine. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow diagrams into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and decision flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

The Using Simulink documentation describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

state

A *state* describes a mode of a reactive system. A reactive system has many possible states. States in a Stateflow diagram represent these modes. The activity or inactivity of the states dynamically changes based on transitions among events and conditions.

Every state has hierarchy. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself. A state also has history that applies to its level of hierarchy in the Stateflow diagram. States can have actions that are executed in a sequence based upon action type. The action types are `entry`, `during`, `exit`, or `on event_name` actions.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

Stateflow block Masked Simulink model that is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add complex event-driven behavior to Simulink simulations. You create models that represent both data and decision flow by combining Stateflow blocks with the standard Simulink and toolbox block libraries. These combined models are simulated using Simulink.

Stateflow Debugger Use to debug and animate your Stateflow diagrams. Each state in the Stateflow diagram simulation is evaluated for overall code coverage. This coverage analysis is done automatically when the target is compiled and built with the debug options. The Debugger can also be used to perform dynamic checking. The Debugger operates on the Stateflow machine.

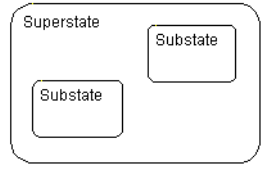
Stateflow diagram Using Stateflow, you create Stateflow diagrams. A *Stateflow diagram* is also a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. See "Stateflow and Simulink" on page 2-5 for more information on Stateflow diagrams.

Stateflow Explorer Use to add, remove, and modify data, event, and target objects. See "The Stateflow Explorer Tool" on page 10-3 for more information.

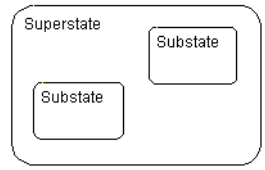
Stateflow Finder Use to display a list of objects based on search criteria you specify. You can directly access the properties dialog box of any object in the search output display by clicking that object. See "The Stateflow Finder Tool" on page 10-31 for more information.

subchart Chart contained by another chart. See "Using Graphical Functions in Stateflow Charts" on page 5-48.

substate A state is a *substate* if it is contained by a superstate.



superstate A state is a *superstate* if it contains other states, called substates.



supertransition Transition between objects residing in different subcharts. See "Using Supertransitions in Stateflow Charts" on page 5-71 for more information.

Target An executable program built from code generated by Stateflow or the Real-Time Workshop. See Chapter 11, "Building Targets," for more information.

topdown processing The way in which Stateflow processes states and events. In particular, Stateflow processes superstates before states. Stateflow processes a state only if its superstate is activated first.

transition The circumstances under which the system moves from one state to another. Either end of a transition can be attached to a source and a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. It is often the occurrence of some event that causes a transition to take place.

transition path Flow path that starts and ends on a state.

transition segment A state-to-junction, junction-to-junction, or junction-to-state part of a complete state-to-state transition. Transition segments are sometimes loosely referred to as transitions.

virtual scrollbar Enables you to set a value by scrolling through a list of choices. When you move the mouse over a menu item with a virtual scrollbar, the cursor changes to a line with a double arrowhead. Virtual scrollbars are either vertical or horizontal. The direction is indicated by the positioning of the arrowheads. Drag the mouse either horizontally or vertically to change the value.

See "Viewing Data and Events from the Editor" on page 5-15 for more information.

Index

A

abs in action language 7-42
accessing existing objects (API)
  with the find method 13-26
  with the findDeep method 13-27
  with the findShallow method 13-27
acos in action language 7-42
action language
  array arguments 7-60
  assignment operations 7-11
  binary operations 7-7
  bit operations 7-7
  categories by example 3-43
  comments 7-76
  condition statements 7-67
  continuation symbols 7-76
  data and event arguments 7-58
  defined 2-18, 3-42
  directed event broadcasting 7-64
  event broadcasting 7-62
  floating-point number precision 7-77
  keyword identifiers 7-4
  literals 7-76
  notation 3-43
  overview 3-42
  pointer and address operations 7-12
  semantics 3-44
  semicolon symbol 7-77
  special symbols 7-76
  temporal logic 7-68
  time symbol 7-76
  types of 7-3
  unary operations 7-10, 7-11
action table in truth tables 9-20
actions
  defined 2-18
  during 3-10
  entry 3-10
  exit 3-10
  on *event_name* 3-10
  states 5-24
  tracking rows in truth tables 9-31
  unary 7-11
  See also condition actions
  See also transition actions
activation order for parallel (AND) states 5-21
active chart execution 4-5
active states 3-9
  display in debugger 12-9
  execution 4-15
  exiting 4-15
addition (+) of fixed-point data 7-29
addition operator (+) 7-8
after temporal logic operator 7-70
animation controls in debugger 12-9
API
  See Stateflow API
Append symbol names with parent names coder option 11-17
arguments 7-58
array arguments in action language 7-60
Array property of data 6-21
arrays
  and custom code 7-61
  defined 6-24
  indexing 7-60
arrowhead size of transitions 5-33
asin in action language 7-42
assignment operations 7-11
  fixed-point data 7-26, 7-32
at temporal logic operator 7-72
atan in action language 7-42
atan2 in action language 7-42

B

Back To button in diagram editor 5-70
`BadIntersection` property (API) 13-24
`before` temporal logic operator 7-71
behavioral properties and methods (API) 14-13
between cells of truth table 9-40
bias (B) in fixed-point data 7-15
binary operations 7-7
  fixed-point data 7-24
binary point in fixed-point data 7-17
bit operations 7-7
bitwise & (AND) operator 7-9
block
  See also Stateflow block
bowing transitions 5-41
Box object (API)
  description 13-6
  methods 15-31
  properties 15-29
boxes
  creating 5-47
  definition 3-40
  grouping 5-47
Break button on debugger 12-8
breakpoints
  calling truth tables 9-35
  chart entry 12-7
  display in debugger 12-9
  event broadcast 12-7
  graphical functions 5-55
  overview 12-3
  setting in debugger 12-7
  state entry 12-7
  states 5-23
  transitions 5-45
broadcasting directed events
  examples using `send` keyword 7-65
  `send` function 4-77
  with qualified event names 4-79
broadcasting events 7-62
  in condition actions 4-32
  in truth tables 9-16
Browse Data display in debugger 12-9
`build` method (API) 16-6
build tools for targets 11-6
building targets 11-3, 11-5
  options for 11-10
  starting 11-25

C

C functions
  library 7-42
  user-written 7-43
Call Stack display in debugger 12-9
cast operation 7-13
`ceil` in action language 7-42
change indicator (*) in title bar 5-7
`change(data_name)` keyword 7-5
chart notes
  See notes (chart)
chart libraries 5-84
Chart object (API)
  accessing 13-10
  create new objects in 13-11
  methods 15-22
  open 13-11
  properties 15-16
charts
  checking for errors 5-83
  creating 5-3
  decomposition 3-9
  editing 5-6, 10-9
  executing active charts 4-5

executing inactive charts 4-5
how they execute 4-5
input data from Simulink 8-8
input events from Simulink 8-6
output data to Simulink 8-9
output events to Simulink 8-7
parent property 10-13
printing 5-85
properties 5-78
saving model 5-3
setting properties for in Explorer 10-9
trigger types 6-11
update method 5-3
update method property 10-13
update methods for defining interface 8-4
See also subcharts
`classhandle` method (API) 16-7
Clipboard object (API)
connecting to 13-33
copying 13-29
description 13-6
methods 15-8
code generation
error messages 11-35
options for 11-11
Code Generation Directory option 11-22
code generation files 11-37
`.dll` files 11-37
code files 11-38
make files 11-39
colors in diagram editor 5-10
comment symbols (%, //, /) in action language 7-76
comments (chart)
See notes (chart)
Comments in generated code coder option 11-13
Compace nested if-else using logical AND/OR operators coder option 11-14
comparison operators
(>, <, >=, <=, ==, -=, !=, <>) 7-8
compilation error messages 11-36
condition actions
and transition actions 4-31
event broadcasts in 4-72
examples 4-29
in `for` loops 4-32
simple, example of 4-29
to broadcast events 4-32
with cyclic behavior to avoid 4-33
condition coverage 12-32
definition 12-32
example 12-40
truth tables 9-66
Condition Table in truth tables 9-20
conditional notation for temporal logic operators 7-74
conditions
for transitions, defined 2-16
for transitions, guidelines 7-67
labels for in truth tables 9-23
outcomes for in truth tables 9-2
configuring targets 11-7, 11-9
conflicting transitions
definition 12-18
detecting 12-18
example 12-18
connecting to
Clipboard object (API) 13-33
Editor object (API) 13-33
Stateflow objects (API) 13-22
connective junctions 3-31
backtracking transition segments to source 4-61

common events example 3-37
common source example 3-36
creating 5-57
definition 3-31
described 2-19
examples of 4-50
flow diagrams 4-55
`for` loop 3-34
`if-then-else` decision 4-51
in flow diagrams 3-32
in `for` loops 4-54
in self-loop transitions 3-34
self-loop transitions 4-53
transitions based on common event 4-60
transitions from a common source 4-57
transitions from multiple sources 4-59
with default transitions 4-36
constructor for Stateflow objects (API) 13-22
containment of Stateflow objects 13-24
Contains word option in Search & Replace tool 10-19
context (shortcut) menu to properties 5-10
context-sensitive constants in fixed-point data 7-23
continuation symbol ... in action language 7-76
Continue button on debugger 12-8
continuous update method for Stateflow block 8-4
`copy` method (API) 16-8
features and limitations 13-30
copying objects (API)
by grouping (recommended) 13-30
`copy` method 13-30
Data, Event, and Target objects 13-31
individual objects 13-31
overview 13-29
using the Clipboard object 13-29
copying objects in the diagram editor 5-14
corners of states 5-29
`cos` in action language 7-42
`cosh` in action language 7-42
Coverage display in debugger 12-9
create (API)
handle to Stateflow objects (API) 13-22
new model and chart (API) 13-9
new objects in chart (API) 13-11
new state (API) 13-11
object containment 13-24
Stateflow objects (API) 13-22
transition (API) 13-12
Creation Date property of machines 10-13
custom code
building into target 11-4, 11-20
building into targets 11-3
calling graphical functions 11-23
integrating with diagram 11-20
path names 11-22
Custom code included at the top of generated code option 11-21
Custom include directory paths option 11-21
Custom initialization code option 11-22
Custom source files option 11-21
custom targets
code generation options 11-17
Custom termination code option 11-22
cutting objects in diagram editor 5-13
cyclic behavior
debugging 12-22
definition 12-22
example 12-23
example of nondetection 12-24
in condition actions 4-33
noncyclic behavior flagged as cyclic example 12-25

cyclomatic complexity
in model coverage reports 12-27

D

dashed transitions 5-32
data 6-26
adding (creating) 6-14
arrays, defining 6-24
associating with ports 6-27
copying/moving in Explorer 10-10
defined 2-15
deleting 10-12
dictionary 3-5
exported 8-27
exporting to external code 6-28
fixed-point 7-14
imported 8-29
importing from external code 6-29
in truth tables 9-32
input from other blocks 6-26
input from Simulink 8-8
operations in action language 7-7
output to other blocks 6-27
output to Simulink 8-9
properties of 6-16
range violations 12-20
renaming 10-10
setting properties for in Explorer 10-9
temporary data 6-28
transferring properties 10-14
viewing 5-15
See also fixed-point data
data and events 8-3
data dictionary
adding data 6-14
adding events 6-2
defined 2-6
data input from Simulink port order 10-11
Data object (API)
methods 15-55
properties 15-50
data output to Simulink port order 10-11
data range violations (debugging) 12-20
debugger
action control buttons 12-7
active states display 12-9
animation controls 12-9
Break button 12-8
breakpoints 12-3
breakpoints display 12-9
browse data display 12-9
call stack display 12-9
clear output display 12-9
Continue button 12-8
coverage display 12-9
debugging run-time errors 12-11
display controls 12-9
error checking options 12-8
including error checking in the target build 12-3
main window 12-5
overview 12-2
setting global breakpoints 12-7
Start button 12-7
status display area 12-7
Step button 12-8
Stop Simulation button 12-8
typical tasks 12-2
user interface 12-5
Debugger breakpoint property
charts 5-81
debugging
conflicting transitions 12-18

cyclic behavior 12-22
data range violations 12-20
model coverage 12-26
state inconsistency 12-16
truth table during simulation 9-56
truth table example 9-57
truth table example model 9-60
truth tables 9-56
decision coverage 12-28
chart as a triggered block 12-29
chart containing substates 12-29
conditional transitions 12-32
example 12-40
in model coverage reports 12-28
state with `on` *`event_name`* statement 12-32
superstates containing substates 12-30
truth tables 9-66
decision outcomes for truth tables 9-2
action labels 9-26
entering 9-25
entering actions for 9-26
entering default decision outcomes 9-26
tracking action rows feature 9-31
decomposition
described 2-11
states and charts 3-9
substates 5-21
default decision outcome for truth tables
concept 9-2
default decision outcomes in truth tables
specifying 9-26
default transitions
and exclusive (OR) decomposition 4-35
and history junctions 4-37
creating 5-34
creating in API 13-35
defined 2-14
examples 3-27, 4-35
labeled 4-39
labeling 3-27
to a junction 4-36
`defaultTransitions` method (API) 16-9
`delete` method (API) 16-10, 16-12
example 13-25
deployment properties and methods (API) 14-20
Description property
data 6-24
events 6-7
graphical functions 5-56
junctions 5-59
machines 10-13
states 5-23
transitions 5-46
truth tables 9-36
Description property for charts 5-81
Destination property of transitions 5-45
destroying Stateflow objects (API) 13-25
Details sections of model coverage report 12-35
diagram (Stateflow)
graphical components 2-11
objects 2-10
saving 1-26
diagram editor
copying objects 5-14
cutting and pasting objects 5-13
drawing area 5-8
elements 5-6
menu bar 5-7
selecting and deselecting objects 5-13
specifying colors and fonts 5-10
status bar 5-8
title bar 5-7
toolbar 5-7
undoing and redoing operations 5-16

zooming 5-15
dialog method (API) 16-11
directed event broadcasting
examples 4-77
send function
examples 7-65
semantics 4-77
using qualified event names 4-79
with qualified names 7-64
disp method (API) 16-12
display controls in debugger 12-9
display in MATLAB symbol (;) 7-77
displaying
enumerated values for properties (API) 13-21
properties and methods (API) 13-19, 13-20
subproperties (API) 13-20
division (/) of fixed-point data 7-30
division operator (/) 7-7
Document Link property
charts 5-81
data 6-24
events 6-7
graphical functions 5-56
junctions 5-59
machines 10-13
states 5-23
transitions 5-46
truth tables 9-36
dot (.) notation (API)
nesting 13-18
drawing area
in diagram editor 5-8
drawing objects in diagram editor 5-8
drawing tools 5-8
during action 3-10
example 3-13

E

E (binary point) in fixed-point data 7-17
early return logic for event broadcasts 4-18
Echo expressions without semicolons coder option 11-12
Edit property of Search & Replace tool 10-25
editing
charts 5-6
labels in diagram editor 5-14
truth tables 9-20
Editor object (API)
connecting to 13-33
description 13-6
graphical changes 13-33
methods (API) 15-7
properties 15-6
Editor property for charts 5-81
either edge trigger 6-11
Enable C-like bit operations property
for charts 5-80
of machines 10-13
operations affected 7-11
Enable debugging/animation coder option 11-12
Enable overflow detection (with debugging) coder option 11-12
entry action 3-10
example 3-13
error checking
charts 5-83
errors in truth tables 9-49
in truth tables 9-48
overspecified truth tables 9-52
underspecified truth tables 9-53
warnings in truth tables 9-51
when it occurs for truth tables 9-48
error messages
code generation 11-35

compilation 11-36
overview 11-34
parsing 11-34
target building 11-36
errors
data range 12-8
debugging run-time errors 12-11
detect cycles 12-8
in truth tables 9-49
state inconsistency 12-8
transition conflict 12-8
event
in truth tables 9-32
event actions
in a superstate 4-63
event broadcasting
early return logic 4-18
examples
state action notation 7-62
transition action notation 7-63
in condition actions 4-72
in parallel state action 4-65
nested in transition actions 4-69
See also directed event broadcasting
event input from Simulink
port order 10-11
trigger 8-7
event notation for temporal logic operators 7-74
Event object (API)
methods 15-58
properties 15-56
event output to Simulink port order 10-11
event triggers
defining 8-19
function call example 8-16
function call output event 8-16
function call semantics 8-18
events 6-2
adding (creating) 6-2
and transitions from substate to substate 4-27
broadcast in condition actions 4-32
broadcasting 7-62
causing transitions 4-24
copying/moving in Explorer 10-10
defined 2-15
defining edge-triggered output events 8-19
deleting 10-12
executing 4-3
exported 8-23
exporting events example 8-23
exporting to external code 6-9
function call output event to Simulink 8-16
how Stateflow processes them 4-4
imported 8-25
imported event example 8-25
importing from external code 6-10
input from Simulink 8-6
local 6-8
output to Simulink 8-7
processing with inner transition to junction 4-45
processing with inner transitions in exclusive (OR) states 4-41
properties 6-4
renaming 10-10
setting properties for in Explorer 10-9
sources for 4-3
transferring properties 10-14
triggering Simulink blocks with 6-9
viewing 5-15
See also directed event broadcasting
See also implicit events
See also input events
See also output events

every temporal logic operator 7-73
exclusive (OR) decomposition 3-9
  and default transitions 4-35
exclusive (OR) states
  defined 2-11
  transitions 3-18
  transitions to and from 4-23
exclusive (OR) substates
  transitions 3-20
exclusive (OR) superstates
  transitions 3-19
Execute (enter) Chart at Initialization property for charts 5-81
exit action 3-10
  example 3-13
exp in action language 7-42
Explore property of Search & Replace tool 10-25
Explorer
  contents of list 10-4
  object hierarchy list 10-4
  object parentage 10-5
  object properties displayed 10-5
  objects by icon 10-5
  opening 10-3
  opening new or existing model 10-9
  operations 10-8
  overview 10-3
  parents of objects in Contents pane 10-5
  parts of Explorer window 10-4
  properties in Contents pane 10-6
  targets 10-7
  user interface 10-3
Export Chart Level Graphical Functions property for charts 5-80
exporting
  truth tables to HTML 9-47
exporting data to external code 8-27
  description 6-28
  example 8-27
exporting events to external code 8-23
  example 8-24
exporting graphical functions 5-53
external code sources
  defining interface for 8-23
  definition 8-23

F

F (fractional slope) in fixed-point data 7-17
F symbol in action language 7-77
fabs in action language 7-42
falling edge trigger 6-11
Field types field of Search & Replace tool 10-18
final action 9-4
final action for truth table 9-4
final action in truth tables 9-29
find method (API) 16-13
  how to use 13-26
find method (API)
  examples 13-9, 13-10
findDeep method (API) 16-16
  how to use 13-27
Finder
  dialog box 10-32
  user interface 10-31
findShallow method (API) 16-17
  how to use 13-27
finite state machine
  described 2-2
  introduction 2-2
  references 2-4
  representations 2-2
First Index (of array) property of data 6-22
fixed-point data 7-14

arithmetic 7-15
bias B 7-15
context-sensitive constants 7-23
defined 7-15
example of using 7-38
how to use 7-20
implementation in Stateflow 7-17
off-line conversions 7-22
on-line conversions 7-22
operation (+, -, *, /) equations 7-16
operations supported 7-24
overflow detection 7-37
properties 6-20
quantized integer, Q 7-15
Scaling property 6-21, 7-19
setting for Strong Data Typing with Simulink IO 5-81
sharing with Simulink 7-37
slope S 7-15
specifying in Stateflow 7-19
Stored Integer property 6-20, 7-19
Type property 7-19
fixed-point operations 7-24
assignment 7-32
casting 7-32
logical (&, &&, |, ||) 7-31
promotions 7-27
special assignment
and context-sensitive constants 7-36
division example 7-35
multiplication example 7-33
floating-point numbers
precision in action language 7-77
`floor` in action language 7-42
flow diagrams
connective junctions in 3-32
cyclic behavior example 12-24
example 3-35
examples 3-32
`for` loops 3-34
with connective junctions 4-55
flow graphs
order of execution 4-7
types 4-6
`fmod` in action language 7-42
font size of labels 5-14
fonts in diagram editor 5-10
`for` loops
example 3-34
with condition actions 4-32
with connective junctions 4-54
Forward To button in diagram editor 5-70
function call events
example output event semantics 8-18
output event 8-16
output event example 8-16
Function Inline Option property
graphical functions 5-56
truth tables 9-36
function notation for API methods 13-18
Function object (API)
description 13-6
methods 15-35
properties 15-33
functions
data and event arguments 7-58
keywords for special functions 7-6
MATLAB 7-46
truth table function 9-2
See also graphical functions

G

`generate` method (API) 16-18

generated code files 11-37
`get` method (API) 16-19
  examples 13-19, 13-20
  getting and setting properties of objects 13-28
`getCodeFlag` method (API) 16-20
graphical functions 3-41
  calling from action language 5-53
  calling from custom code 11-23
  compared with truth tables 9-16
  creating 5-48
  example 3-41
  exporting 5-53
  inlining 5-56
  properties 5-54
  realizing truth tables 9-68
  signature (label) 5-48
graphical objects 3-3
  copying 5-14
  cutting and pasting 5-13
  drawing in diagram editor 5-8
graphical properties and methods (API) 14-30
grouping
  boxes 5-47
  states 5-20

H

`help` method (API) 16-21
  example 13-19, 13-20
hexadecimal notation in action language 7-77
hierarchy
  described 2-21
  of objects 3-5
  of states 3-8
  state example 3-6
  states 3-5
  transition example 3-7
history junctions 3-38
  and default transitions 4-37
  and inner transitions 3-39
  creating 5-57
  defined 2-17
  definition 3-38
  example of use 3-38
  inner transitions to 3-25, 4-48

I

`if-then-else` decision
  examples 3-32, 3-33
  with connective junctions 4-51
implicit events
  definition 6-12
  example 6-12
  keyword identifiers 7-5
  referencing in action language 6-12
importing data from external code 6-29, 8-29
  example 8-29
importing events from external code 8-25
  example 8-26
inactive chart execution 4-5
inactive states 3-9
Index property for events 6-7
inherited update method for Stateflow block 8-4
initial action for truth tables 9-4
initial action in truth tables 9-29
Initialize from property of data 6-22
inlining
  truth table functions 9-36
inlining graphical functions 5-56
inner transitions
  after using them 3-24
  before using them 3-23
  definition 3-22

examples 3-22, 4-41
processing events in exclusive (OR) states 4-41
to a history junction 4-48
to a junction, processing events with 4-45
to history junction 3-25
`innerTransitions` method (API) 16-22
input data from other blocks 6-26
input events
associating with control signals 6-8
defining
integer word size
setting for target 7-28
interfaces 8-3
to external code 8-2, 8-23
to MATLAB data 8-2
typical tasks to define 8-3
update methods for Stateflow block 8-4
interfaces to MATLAB
data 8-22
workspace 8-22
interfaces to Simulink
continuous Stateflow block 8-14
defining 2-7
edge-triggered output event 8-19
function call output event 8-16
implementing 8-11
inherited Stateflow block 8-13
sampled Stateflow block 8-12
triggered Stateflow block 8-11

J

Junction object (API)
properties 15-48
junctions
moving 5-58
properties 5-58
size 5-58
See also connective junctions
See also history junctions

K

keyboard navigation 9-40
in truth tables 9-40
keywords
`change(data_name)` 7-5
`during` 7-4
`entry` 7-4
`entry(state_name)` 7-5
`exit` 7-5
`exit(state_name)` 7-5
implicit events 7-5
`in(state_name)` 7-6
`ml()` 7-6
`ml.` 7-6
`on` *`event_name`* action 7-5
`send` 7-6
special functions 7-6
state action language 7-4
summary list 7-4
`tick` 7-5
`wakeup` 7-5

L

Label property
graphical functions 5-56
states 5-23
transitions 5-45
truth tables 9-36
labels
default transitions 3-27, 4-39

editing in diagram editor 5-14
field 10-22
font size 5-14
for actions in truth tables 9-26
format for transition segments 4-50
format for transitions 4-23, 5-30
graphical function signature 5-48
multiline labels using API 13-34
state example 3-12
states 3-10, 5-23
transition 3-15
transitions 5-30
truth table signature 9-18
`labs` in action language 7-42
`ldexp` in action language 7-42
left bit shift (<<) operator 7-8
Limit Range property of data 6-22
listing
enumerated values for properties (API) 13-21
properties and methods (API) 13-19, 13-20
subproperties (API) 13-20
literal code symbol $ in action language 7-76
`log` in action language 7-42
`log10` in action language 7-42
logical AND operator (&) 7-9

M

machine
adding targets to 11-7
overview of Stateflow machine 10-2
setting properties 10-12
Machine object (API)
accessing 13-9
description 13-6
methods 15-14
properties 15-11
make files 11-39
`make` method (API) 16-23
Match case
field of Search & Replace tool 10-18
search option of Search & Replace tool 10-19
Match options field of Search & Replace tool 10-18
Match whole word option in Search & Replace tool 10-20
MATLAB
API scripts 13-38
display symbol (;) in action language 7-77
functions and data in Stateflow 7-46
`ml()` and full MATLAB notation 7-50
`ml()` function call 7-47
`ml.` namespace operator 7-46
See also interfaces to MATLAB
See also interfaces to MATLAB workspace
`max` in action language 7-42
Max property of data 6-22
MCDC coverage
definition 12-33
example 12-40
explanation 12-42
irrelevant conditions 12-43
specifying 12-27
truth tables 9-66
menu bar
in diagram editor 5-7
messages
error messages 11-34
of Search & Replace tool 10-28
methods (API)
description of 13-7
displaying 13-19, 13-20
function notation 13-18
naming 13-17

nesting 13-18
of Box object 15-31
of Chart object 15-22
of Clipboard object 15-8
of Data object 15-55
of Editor object 15-7
of Event object 15-58
of Function object 15-35
of Machine object 15-14
of Note object 15-43
of State object 15-27
of Transition object 15-47
of Truth Table object 15-40
overview of methods 16-2
`methods` method (API) 16-24
example 13-19, 13-20
`min` in action language 7-42
Min property of data 6-22
Minimize array reads using temporary variables coder option 11-16
`ml` data type 7-51
and targets 7-51
inferring size 7-51
place holder for workspace data 7-52
scope 7-51
`ml()` function 7-47
and full MATLAB notation 7-50
dynamically construct workspace variables 7-50
expressions 7-49
inferring return size 7-53
or `ml.` namespace operator, which to use? 7-50
`ml.` namespace operator 7-46
expressions 7-49
inferring return size 7-53
or `ml()` function, which to use? 7-50
model
opening new or existing in Explorer 10-9
model coverage 12-26
chart as subsystem report section 12-36
condition coverage 12-32
coverages for truth table function 9-66
cyclomatic complexity 12-27
decision coverage 12-28
definition 12-26
for Stateflow charts 12-33
for truth tables 9-64
generate HTML report 12-27
MCDC coverage 12-33
report 12-26
report for truth table example 9-64
reporting on 12-26
specifying reports 12-27
truth tables 9-64
model coverage report
chart as superstate section 12-37
Details sections 12-35
state sections 12-38
Summary 12-34
transition section 12-40
modulus operator (%%) 7-8
ms>.dll files 11-37
multiplication (*) of fixed-point data 7-30
multiplication operator (*) 7-7

N

Name property
charts 5-79
data 6-17
events 6-5
graphical functions 5-55
states 3-12, 5-22

truth tables 9-35
naming of properties and methods (API) 13-17
No Code Generation for Custom Targets property for charts 5-80
nongraphical objects (data, events, targets) 3-4
nonsmart transitions
asymmetric distortion 5-44
graphical behavior
notation
defined 2-3
introduction to Stateflow notation 3-1
representing hierarchy 3-5
Note object (API)
methods 15-43
properties (API) 15-41
notes (chart)
changing color 5-62
changing font 5-62
creating 5-61
deleting 5-63
editing existing notes 5-62
moving 5-63
TeX format 5-62

O

object palette
in Stateflow diagram editor 5-8
Object types field of Search & Replace tool 10-18
objects
hierarchy 3-5
overview of Stateflow objects 3-3
See also graphical objects
See also nongraphical objects
objects (API)
copying 13-29
getting and setting properties 13-28
off-line conversions with fixed-point data 7-22
`on event_name` action 3-10
example 3-13
on-line conversions with and fixed-point data 7-22
operations
assignment 7-11
binary 7-7
bit 7-7
cast 7-13
defined for fixed-point data 7-16
exceptions to undo 5-17
fixed-point data 7-24
in action language 7-7
pointer and address 7-12
typecast 7-13
unary 7-10
undo and redo 5-16
with objects in Explorer 10-8
operators
addition (+) 7-8
bitwise AND (&) 7-9
bitwise OR (|) 7-10
bitwise XOR (^) 7-9
comparison (>, <, >=, <=, ==, -=, !=, <>) 7-8
division (/) 7-7
left bit shift (<<) 7-8
logical AND (&&) 7-10
logical AND (&) 7-9
logical OR (|) 7-10
logical OR (||) 7-10
modulus (%%) 7-8
multiplication (*) 7-7
pointer and address 7-12
power (^) 7-9
right bit shift (>>) 7-8
subtraction (-) 7-8

outerTransitions method (API) 16-25
output data to other blocks 6-27
output events
  associating with output port 6-9
  defining 6-9
Output State Activity property of states 5-22
outputData method (API) 16-26
overflow detection
  fixed-point data 7-37
overlapping object edges 13-24
overspecified truth tables 9-52
overview of API methods 16-2

P

parallel (AND) states
  activation order 5-21
  decomposition 3-10
  defined 2-11
  entry execution 4-13
  event broadcast action 4-65
  examples of 4-65
  order of execution 4-13
Parent property
  charts 5-79
  data 6-17
  events 6-6
  graphical functions 5-55
  junctions 5-59
  states 5-23
  transitions 5-45
  truth tables 9-35
parse method (API) 16-28
parsing diagrams
  error messages 11-34
  example 11-28
  overview 11-27
  starting the parser 11-27
  tasks 11-28
passing arguments by reference
  C functions
    passing arguments by reference 7-45
pasteTo method (API) 16-29
pasting objects in the diagram editor 5-13
path names for custom code 11-22
pointer and address operations 7-12
Port property
  data 6-21
  events 6-7
ports
  association with data 6-27
  order of inputs and outputs 10-11
pow in action language 7-42
Preserve case
  field of Search & Replace tool 10-18
  search type in Search & Replace tool 10-21
Preserve symbol names coder option 11-17
printing
  book report of elements 5-90
  charts 5-85
  current diagram 5-90
  details of chart 5-87
  diagram 5-85
  truth tables 9-47
promotion rules for fixed-point operations 7-27
properties
  machine 10-12
  of truth tables 9-34
  Search & Replace tool 10-25
  states 5-22
  transferring between data, events, and targets 10-14
properties (API)
  description of 13-7

displaying 13-19, 13-20
displaying enumerated values for 13-21
displaying subproperties 13-20
getting and setting 13-28
naming 13-17
nesting 13-18
of Box object 15-29
of Chart object 15-16
of Data object 15-50
of Editor object 15-6
of Event object 15-56
of Function object 15-33
of Junction object 15-48
of Machine object 15-11
of Note object 15-41
of State object 15-24
of Target object 15-59
of Transition object 15-44
of Truth Table object 15-37
properties and methods (API)
behavioral 14-13
deployment 14-20
graphical 14-30
structural 14-5
utility and convenience 14-26
Properties property of Search & Replace tool 10-25

Q

quantized integer (Q) in fixed-point data 7-15
Quick Start
creating a Simulink model 1-8
creating a Stateflow diagram 1-10
debugging the Stateflow diagram 1-25
defining the Stateflow interface 1-15
generating code 1-24
overview 1-8
running a simulation 1-21
Stateflow API 13-9
Stateflow typical tasks 1-8

R

`rand` in action language 7-42
range violations, data 12-20
`rebuildAll` method (API) 16-30
Recognize if-elseif-else in nested if-else statements coder option 11-15
redo operation 5-16
references 2-4
`regenerateAll` method (API) 16-31
regular expressions
Search & Replace tool 10-20
Stateflow Finder 10-33
tokens in Search & Replace tool 10-20
relational operations
fixed-point data 7-30
renaming targets 11-8
Replace button of Search & Replace tool 10-18
replace buttons in Search & Replace tool 10-27
Replace constant expressions by a single constant coder option 11-15
Replace with field of Search & Replace tool 10-18
replacing text in Search & Replace tool 10-26
with case preservation 10-26
with tokens 10-27
reports
book report of elements 5-90
charts 5-85
details of chart 5-87
model coverage 12-26
model coverage for Stateflow charts 12-33
resolving symbols in action language 11-32

return size of ml expressions 7-53
right bit shift (>>) operator 7-8
rising edge trigger 6-11
Root object (API)
  access 13-9
  description 13-5
rtw target
  code generation options 11-13
  starting the build 11-25
run-time errors
  debugging 12-11

S

Sample Time property for charts 5-79
sampled update method for Stateflow block 8-4
Save final value to base workspace property of data 6-24
saving
  Simulink model (API) 13-16
  Stateflow diagram 1-26
Scaling property of fixed-point data 6-21, 7-19
Scope property
  data 6-18
  events 6-6
script of API commands 13-38
Search & Replace tool 10-16
  containing object 10-24
  Contains word option 10-19
  Custom Code field 10-23
  Description field 10-23
  Document Links field 10-23
  Field types field 10-18
  icon of found object 10-24
  Match case field 10-18
  Match case option 10-19
  Match options field 10-18
  Match whole word option 10-20
  messages 10-28
  Name field 10-22
  object types 10-18
  Object types field 10-18
  opening 10-16
  portal area 10-24
  Preserve case field 10-18
  Preserve case option 10-21
  Regular expression option in Search & Replace tool 10-20
  regular expression tokens 10-20
  Replace All button 10-28
  Replace All in This Object button 10-28
  Replace button 10-18, 10-27
  Replace with field 10-18
  replacement text 10-26
  Search button 10-18, 10-23
  Search For field 10-17
  Search in field 10-18
  search order 10-24
  search scope 10-21
  search types 10-19
  view area 10-23
  View Area field 10-19
  viewer 10-24
  viewing a match 10-24
Search button of Search & Replace tool 10-18
Search for field of Search & Replace tool 10-17
Search in field of Search & Replace tool 10-18
search order in Search & Replace tool 10-24
search scope in Search & Replace tool 10-21
searching
  chart 10-21
  Finder user interface 10-31
  machine 10-21
  specific objects 10-22

text 10-16
text matches 10-22
selecting and deselecting objects in the diagram editor 5-13
self-loop transitions 3-22
creating 5-33
delay 3-35
with connective junctions 4-53
with junctions 3-34
semantics
defined 2-4
early return logic for event broadcasts 4-18
examples 4-21
executing a chart 4-5
executing a state 4-13
executing a transition 4-6
executing an event 4-3
semicolon 7-77
`send` function
and directed event broadcasting 7-64
directed event broadcasting 4-77
directed event broadcasting examples 7-65
`set` method (API) 16-32
`setCodeFlag` method (API) 16-33
`sfclipboard` method (API) 16-34
example 13-33
`sfexit` method (API) 16-35
`sfhelp` method (API) 16-36, 16-42
`sfnew` function 5-3
`sfnew` method (API) 16-37
`sfprint` method (API) 16-38
`sfprj` directory
generated code 1-24
model information 1-26
`sfsave` method (API) 16-43, 16-44
shortcut menus
in Stateflow diagram editor 5-8
to properties 5-10
Show Portal property of Search & Replace tool 10-25
signature
graphical functions 5-48
truth tables 9-18
simulating truth tables 9-56
simulation target
code generation options 11-11
starting the build 11-25
Simulink
truth tables example (first) 9-7
See also interfaces to Simulink
Simulink model and Stateflow machine
relationship between 2-5
Simulink Model property of machines 10-13
Simulink Subsystem property for charts 5-79
`sin` in action language 7-42
single-precision floating-point symbol F 7-77
`sinh` in action language 7-42
Sizes (of array) property of data 6-21
slits (in supertransitions) 5-71
slope (S) in fixed-point data 7-15
smart transitions
bowing symmetrically 5-41
graphical behavior 5-35
Source property of transitions 5-45
`sourcedTransitions` method (API) 16-46
`sqrt` in action language 7-42
Start button on debugger 12-7
starting the build 11-25
state inconsistency
debugging 12-16
definition 12-16
detecting 12-16
example 12-17
State object (API)

description 13-6
methods 15-27
properties 15-24
Stateflow
applications, types of 1-27
component overview 2-9
design approaches 1-27
feature overview 1-2
representations 2-3
Stateflow API
Box object 13-6
Chart object (API), accessing 13-10
Clipboard object 13-6
common properties and methods 13-7
create new model and chart 13-9
Editor object (API) 13-6
Function object 13-6
Machine object 13-6
Machine object (API), access 13-9
methods of objects 13-7
naming and notation 13-17
object hierarchy 13-4
open chart 13-11
overview 13-3
properties of objects 13-7
Quick Start 13-9
references to properties and methods 13-7
Root object 13-5, 13-9
State object 13-6
unique properties and methods 13-7
Stateflow blocks
considerations in choosing continuous update 8-14
continuous 8-5, 8-14
continuous example 8-15
inherited 8-4, 8-13
inherited example 8-13
sampled 8-5, 8-12
sampled example 8-12
triggered 8-4, 8-11
triggered example 8-11
update methods 8-4
Stateflow diagram editor
object palette 5-8
shortcut menus 5-8
zoom control 5-8
`stateflow` function 5-3
`stateflow` method (API) 16-47
`Stateflow.State` method (API) 16-49, 16-50, 16-51, 16-52, 16-53, 16-54, 16-55, 16-56, 16-57
states
actions 5-24
active and inactive 3-9
active state execution 4-15
button (drawing) 3-8
corners 5-29
create (API) 13-11
creating 5-18
debugger breakpoint property 5-23
decomposition 3-8, 3-9
definition 3-8
`during` action 3-13
editing 10-9
`entry` action 3-13
entry execution 4-13
exclusive (OR) decomposition 3-9
execution example 4-16
`exit` action 3-13
exiting active states 4-15
grouping 5-20
hierarchy 3-6, 3-8
how they are executed 4-13
keywords for action language 7-4
label 3-10, 5-23, 5-24

label example 3-12
label notation 3-8
label property 5-23
label, multiline (API) 13-34
moving and resizing 5-19
Name property 3-12
Name, entering 5-25
`on` *`event_name`* action 3-13
output activity to Simulink 5-26
parallel (AND) decomposition notation 3-10
properties 5-22
representing hierarchy 3-5
setting properties for in Explorer 10-9
See also parallel states
status bar
in diagram editor 5-8
Step button on debugger 12-8
Stop Simulation button on debugger 12-8
Stored Integer property of fixed-point data 6-20
Strong Data Typing with Simulink IO setting
fixed-point data 5-81
`struct` method (API) 16-58
structural properties and methods (API) 14-5
subcharts
creating 5-64, 5-66
definition and description 5-64
editing contents 5-69
manipulating 5-68
navigating through hierarchy of 5-69
opening to edit contents 5-68
unsubcharting 5-66
and supertransitions 5-64
substates
creating 5-20
decomposition 5-21
subtraction (-) of fixed-point data 7-29
subtraction operator (-) 7-8
Summary of model coverage report 12-34
superstates
event actions in 4-63
supertransitions 5-71
creating (drawing) 5-72
definition and description 5-71
labeling 5-77
slits 5-71
working with in the API 13-36
Symbol Autocreation Wizard 11-32
symbols in action language 7-76

T

`tan` in action language 7-42
`tanh` in action language 7-42
Target object (API)
properties 15-59
targets
adding to machine 11-7
build options 11-10
building 11-3
building custom code into 11-20
building error messages 11-36
building procedure 11-5
building with custom code 11-4
code generation options 11-11
configuration options 11-9
configuring 11-7
copying/moving in Explorer 10-10
custom code 11-3
deleting 10-12
in Explorer 10-7
overview 11-3
rebuilding 11-4
renaming 11-8
rtw 1-28

setting integer word size for 7-28
setting properties for in Explorer 10-9
setting up build tools 11-6
transferring properties 10-14
types of 11-3
See also custom targets
See also `rtw` target
See also simulation targets
temporal logic events 7-74
temporal logic operators 7-68
`after` 7-70
`at` operator 7-72
`before` operator 7-71
event notation 7-74
`every` operator 7-73
rules for using 7-68
temporary data 6-28
text
replacing 10-16
searching 10-16
`tick` keyword 7-5
time symbol t in action language 7-76
title bar
in diagram editor 5-7
toolbar
in diagram editor 5-7
transition actions
and condition actions 4-31
event broadcasts nested in 4-69
notation 3-15
transition labels
condition 5-30
condition action 5-30
event 5-30
multiline (API) 13-34
transition action 5-30
Transition object (API)
labels, multiline 13-34
methods 15-47
properties 15-44
transition segments
backtracking to source 4-61
label format 4-50
transitions
and exclusive (OR) states 3-18, 4-23
and exclusive (OR) substates 3-20
and exclusive (OR) superstates 3-19
arrowhead size 5-33
based on events 4-24
bowing 5-32
changing arrowhead size 5-33
condition 5-30
condition action 5-30
connection examples 3-18
create (API) 13-12
creating 5-28
dashed 5-32
debugging conflicting 12-18
default transitions (API) 13-35
defined 2-13
deleting 5-28
events 5-30
flow graph types 4-6
from common source with connective junctions 4-57
from connective junctions based on common event 4-60
from multiple sources with connective junctions 4-59
hierarchy 3-7
label format 5-30
labels
action semantics 4-23
format 5-30

overview 3-15, 5-30
moving 5-31
moving attach points 5-32
moving label 5-32
non-smart
anchored connection points 5-43
notation 3-18
ordering by angular surface position 4-10
ordering by hierarchy 4-9
ordering by label 4-9
ordering single source transitions 4-8
overview 3-14
properties 5-44, 5-45
self-loop transitions 5-33
setting them smart
smart
connecting to junctions at 90 degree angles 5-38
sliding and maintaining shape 5-37
sliding around surfaces 5-36
snapping to an invisible grid 5-40
straight transitions 5-29
substate to substate with events 4-27
supertransitions in the API 13-36
transition action 5-30
valid 3-17
valid labels 5-31
when they are executed 4-6
See also default transitions
See also inner transitions
See also nonsmart transitions
See also self-loop transitions
See also smart transitions
trigger
event input from Simulink 8-7
Trigger property
events 6-7
trigger types for charts 6-11
triggered update method for Stateflow block 8-4
Truth Table object (API)
methods 15-40
properties 15-37
truth tables 9-4
action labels 9-26
appending rows and columns 9-38
calling rules 9-16
changing row height 9-43
checking for errors 9-48
compared with graphical functions 9-16
condition and action tables 9-20
condition labels 9-23
copying and pasting contents of cells 9-41
creating 9-18
data in 9-32
Debugger breakpoinst property 9-35
debugging 9-56
debugging example 9-57
default decision 9-2
defined 9-2
deleting contents of cells 9-40
deleting rows and columns 9-43
Description property 9-36
displaying single table option 9-44
Document Link property 9-36
edit operations overview 9-37
editing 9-20
editing text in cells 9-40
entering actions for decision outcomes 9-26
entering conditions 9-23
entering decision outcomes 9-25
entering default decision outcomes 9-26
entering final actions 9-29
entering initial actions 9-29
events in 9-32

example first truth table 9-7
exporting to HTML 9-47
how they are realized 9-68
how to interpret 9-2
initial action 9-4
inlining 9-36
inserting rows and columns 9-39
keyboard navigation 9-40
keyboard navigation between cells 9-40
Label (signature) property 9-36
model coverage 9-64
model coverage example report 9-66
model coverage for 9-64
model for debugging 9-60
moving rows and columns 9-42
Name property 9-35
overspecified 9-52
Parent Property 9-35
printing 9-47
properties dialog 9-34
pseudocode example 9-2
row and column tooltips 9-44
seeing generated function for 9-68
selecting individual cells 9-39
selecting individual cells for editing 9-39
selecting rows and columns 9-42
signature 9-18
simulation 9-56
specification overview 9-18
steps for creating and using 9-4
underspecified 9-53
undo and redo edit operation 9-37
using in Stateflow overview 9-14
when generated 9-68
why use in Stateflow 9-14
Type property
data 6-19
fixed-point data 7-19
typecast operation 7-13
typographical conventions (table) xxxv

U

unary actions 7-11
unary operations 7-10
fixed-point data 7-26
underspecified truth tables 9-53
undo operation 5-16
exceptions 5-17
Units property of data 6-21
Up To button in diagram editor 5-70
Update method property for charts 5-79
update methods for Stateflow block 8-4
Use chart names with no mangling coder option 11-17
Use Strong Data Typing with Simulink I/O property for charts 5-81
user-written code
and Stateflow arrays 7-61
C functions 7-43, 7-45
utility and convenience properties and methods (API) 14-26

V

valid transitions 3-17
Version property of machines 10-13
View Area field of Search & Replace tool 10-19
view area of Search & Replace tool 10-23
`view` method (API) 16-59

W

`wakeup` keyword 7-5

warnings
during error checking 9-51
Watch in debugger property of data 6-24
workspace
examining the MATLAB workspace 8-22
wormhole 5-74

Z

zoom control
in Stateflow diagram editor 5-8
`zoomIn and zoomOut` methods (API) 16-60
zooming a diagram
overview 5-15
shortcut keys 5-16
using zoom factor selector 5-15